Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

A Type System for Access Control in an Object-Oriented Language

Mário Rui Dias Pires (aluno nº 26227)

Julho de 2009

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

A Type System for Access Control in an Object-Oriented Language

Mário Rui Dias Pires (aluno nº 26227)

Orientador: Prof. Dr. Luís Caires

Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.

Julho de 2009

Acknowledgements

I would first like to thank Prof. Caires for his thesis proposal made specially for me and his guidance throughout its development. I thank all my colleagues from the PLT research group for their comments and suggestions on the initial stages of this work.

A special thanks to my friend Bernardo Toninho for his insights and ideas, some of them quite helpful. I thank my family for their support, specially my sister Cláudia, who fed me when I was hungry.

This work was supported by a CITI research grant.

Resumo

A necessidade de garantir a integridade de dados protegidos leva ao desenvolvimento de sistemas de controlo de acesso. Nesta dissertação, desenvolvemos e formalizamos um sistema de tipos e efeitos que verifica o controlo de acessos a objectos numa linguagem orientada a objectos.

Tradicionalmente, o controlo de acesso é feito apenas em tempo de execução utilizando técnicas dinâmicas, como listas de controlo de acesso. No entanto, estas técnicas aumentam o tempo de execução total de uma operação, podendo quebrar requisitos de sistema. Abordagens estáticas, baseadas na análise estática ou sistemas de tipos, reduzem a quantidade de verificações em tempo de execução fazendo-as em tempo de compilação, prevenindo a ocorrência de erros e oferecendo provas formais de correcção.

O sistema de tipo desenvolvido na presente dissertação aborda a delegação dinâmica de autorizações para aceder a objetos. Uma autorização inclui a identificação do objecto protegido e uma política de acesso e é considerada pelo sistema de tipos como um valor de primeira classe. Assim, os tipos para objectos são estendidos com políticas que reflectem os privilégios actuais associados a esse objecto, e tipificar uma expressão pode produzir efeitos sobre essas políticas. A este novo tipo designamos *user type* e o respectivo valor *user view*, que contêm a referência para o objecto e uma respectiva política de acesso.

Consideramos que os privilégios sobre objectos são os métodos que podem ser invocados. Assim, uma política indica quais são os métodos visiveis de um objecto. Quando uma chamada de método é tipificada, é possível verificar se essa chamada foi autorizada, ou seja, se a política corrente diz que o método está visível. Esta abordagem permite remover as especificações de segurança de declarações de classes, como é o caso de modificadores de visibilidade (public, private).

Finalmente, apresentamos um resultado de consistência para o nosso sistema de tipos. Também implementamos um algoritmo de verificação de tipos para o nosso sistema de tipos, resultando numa ferramenta para verificar a integridade de objectos protegidos num sistema implementado na linguagem de programação definida.

Palavras-chave: Segurança, Controlo de Acesso, Sistema de Tipos

Abstract

The need for a security system to ensure the integrity of protected data leads to the development of access control systems, whose purpose is to prevent access to protected information or resources by unauthorized individuals. In this thesis, we develop and formalize a type and effect system that verifies the access control to objects in a simplified object-oriented language.

Traditionally, access control is done only at run-time, using dynamic techniques, such as access control lists, that perform run-time verifications for credentials and privileges. However, these techniques increase the total execution time of an operation, potentially breaking system requirements such as usability or response time.

Static approaches, based on static analysis or type systems, reduce the amount of run-time checks by doing some of those checks during compile-time, preventing the occurrence of errors before running the program and offering formal proofs of system correctness.

The type system developed in this dissertation deals with the dynamic delegation of authorizations to access objects. An *authorization* includes the identification of the protected object and its access policy and is considered by the type system as a first class value. As such, object types are extended with policies that reflect the current privilege associated with the object, and typing an expression can produce an effect on policies. We name this new type as *user type* and the respective value as *user view*, which contain the object's reference and a policy to access the object.

We consider privileges over objects to be the methods that can be invoked. So, a policy states what methods are available to be called. When typing a method call by an user view, we are able to verify if it was authorized, that is, if the current policy says that the method is available. This mechanism allows the removal of common security specifications from class declarations, as visibility modifiers (public, private).

Furthermore, we present a soundness result for our type system. We also implemented a typechecking algorithm for our type system, resulting in a tool to verify the integrity of protected objects in a system designed in the defined programming language.

Keywords: Security, Access Control, Type System

Contents

List of Figures

1. Introduction

The goal of this thesis is the development and formalization of a type system for an object-oriented language in order to statically ensure access control to objects. To this end, we add primitives to a language similar to Java to create and manage authorizations - security tokens that describe a policy to access an object. Authorizations are considered first-class values and can be stored in data structures or exchanged between processes.

Software security is a recent discipline. Only in recent decades computers have become an indispensable tool to businesses and domestic homes. In its early days, computer security was mainly physical protection of hardware, such as security systems of rooms where the servers were placed. On the other end, software owners hired security experts and were depended of their ethical responsibilities to ensure its security. With the marketing of personal computers and network communications, the early concept of security has become obsolete - the exponential increase of system users undermined its stability.

Most system users are reliable - they only use it to get the desired results. However, there are individuals whose goals involve malicious operations, such as theft or destruction of information. At the end of the 80s, Robert T. Morris, a graduate student, created the first *worm*, which spread very rapidly in the newly created Internet [53]. This attack led to a new sub-culture: *hacking*. During the 90s, new attacks were created, more dangerous and complex than the original, exploiting systems vulnerabilities [37]. To face this trend, new security methods have been developed [40], making it one of the most complex system requirements.

Modern computer security is not defined only by consequences of user interaction. Critical systems such as aerial transportation systems or hospital systems have non-interactive components which cannot fail. In these cases, security is a crucial requirement [33]. To carry out such objectives it is necessary to verify if all system processes are safely executing, without the possibility of runtime errors.

All these events led to a rapid evolution of computer security. During the 80s, computer security was an area dedicated to the military, academics and specialists. Nowadays, any developer sees this field as one of the most important in a system development. Any operating system, database manager, *web-service*, or critical application implements a robust and efficient security system, otherwise it would not survive in the current world.

1.1 Motivation

Being established the importance of software security, the question now is how can we implement a mechanism able to deliver such goals.

Firstly, we have to be realistic. A complete fail proof system is very difficult if not impossible to create. The best we can do is to develop tools that help us minimize execution errors, either by creating system models and checking them against security properties or by verifying

code before execution or even by controlling resources during run-time.

Secondly, we have to decide what kind of security we want to address. In section 2.1 we analyze various classes of security. As we will observe, everything from user control to memory management can be considered access control. It all boils down to when and what actions a subject can perform within the system.

Having decided what kind of security we want to ensure, we then need to choose how to implement it. Model driven security is based on model checkers that verify security properties using various logics. Due to its abstract nature, it is perfect to verify security protocols for complex systems such as distributed ones, but lacks the implementational details. Run-time security uses run-time checks to verify if conditions are met to perform the intended action. So, it is best suited for security checks based on values, such as message decryption, but is time consuming. Language-based security, on the other hand, prevents execution errors by using techniques that analyze and verify the code at compile time. Code notations and type systems are often used and are fairly easy to develop and apply. The disadvantages of this approach is the lack of longevity for more complex systems, since these kind of solutions often only perform local reasoning.

A type system is this thesis backbone. One of our objectives is to create a tool easy to use and debug by programmers. To this end, type systems offer simplicity in use and are a common property of modern general purpose languages. Section 2.2 has more information on type systems.

Furthermore, we implement a type system in an object-oriented language similar to Java. System resources are often implemented as objects in this paradigm and therefore verification techniques aimed to objects are extremely useful. Controlling an object behavior (how it is used) with notations indicating if a method can be called may be a powerful yet simple technique for properly certifying the usage of an object.

As a motivating example, in figure 1.1 we have a skeleton of a class *File* with the usual methods open, read, write and close, and two methods that use an object instance of that class.

As a system requirement, we want to be sure that the file is properly used, i.e., first it must be open, then one could read or write and finally close it. With traditional type systems we cannot statically guarantee this requirement. It could be done with dynamic checks using a flag representing the state of the file and verify it each time a method is called to be sure that the file is in the right state. Also, if we want to add permissions for different users to read or write the file, we end up creating more dynamic checks in order to validate the respective calls. If we think that some methods once executed cannot be undone, these validations had to become even more complex, slowing down the method execution time.

As we can see, this simple example raises some issues. The type system developed in this thesis is directed to solve these problems during compile-time. If a program typechecks, it can safely execute with a minimum of run-time verifications to ensure that objects are used in accordance to an authorized policy. To that end, we introduce two new values to the language that, used together, achieve this goal: authorizations and user views.

The novelty of this approach for typing access control is the transition from static access

```
class File {
    public void open() {...}
    public void read() {...}
    public void write() {...}
    public void close() {...}
}

void create(){
    File f = new File();
    DoRead(f);
    DoWrite(f);
}

void DoRead(File f){
    f.open(); f.read(); f.read(); f.close();
}

void DoWrite(File f){
    f.open(); f.write(); f.write(); f.close();
}
```

Figure 1.1 Traditional File class declaration and usage.

control policies defined in classes to mutable access control policies stored on user views and modified by authorizations.

1.2 Proposed Solution - a First Glimpse

We developed a type system for access control of objects in an object-oriented language.

The language is an adaptation of ClassicJava [24], without class inheritance. We extend this simple language with a new value, authorization, and two new primitives to create and use such values. In chapter 4 we formalize the language.

Authorizations are first class values that grant access to an object. They can be stored in structures or passed between processes or simply be an argument for a method call. During runtime, an authorization can be seen as a security token containing the protected object identification and the respective usage policy. Moreover, authorizations are created with cryptographic techniques, being impossible to forge or replicate.

The goal of the type system developed in this work is to verify if the usage of an object

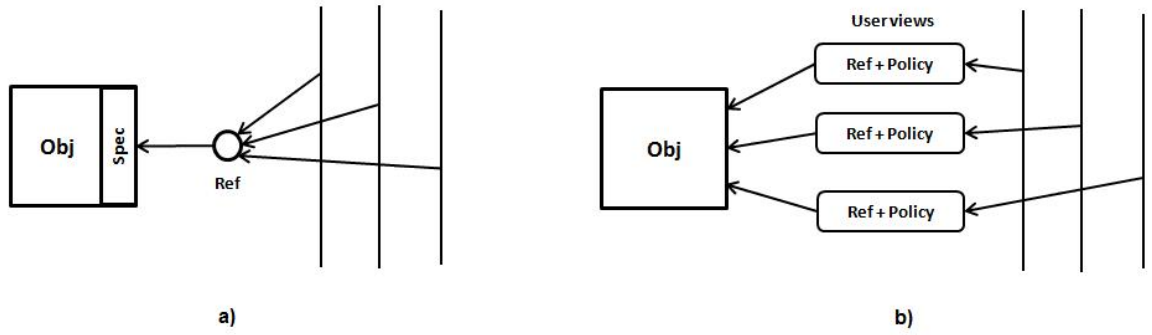

Figure 1.2 Object visibility in a) traditional approaches and b) this work's approach.

complies with the applied authorization, i.e., the object methods are being called according to the policy stored in that authorization. To do so, each object has associated with its type the current policy. One can change such policy with the *authorize* primitive or by calling methods.

A policy can be seen as a view to an object. It defines the methods' visibility and can also define the overall object behavior. So, we add another value to our language: *user views*. Figure 1.2 shows the difference between static object visibility (1.2a) and user views (1.2b).

In the first case, the class is designed as a static interface where one can create a specification to describe the visibility of methods. This specifications can have various forms. The traditional approach is visibility modifiers, as the Java modifiers [28, section 6.6]: private, protected, public and default, that restrict the environment on which a method can be called. Private methods, for example, can only be called within the class they are declared.

In the second case, classes have no specifications; each of its methods can theoretically be called at any time. What restricts where and when a method can be called are user views' policies. A user view is the bridge between whoever wants to access an object (the user) and the object itself, restricting its usage with a policy. Each user view is only used by one user and objects can only be accessed through a user view. Authorizations are used to change such policies and grant access to the object. In chapter 3 we will further describe the concepts introduced here, as well as give examples that apply the proposed solution and solve this work's main challenges.

The previous example is now rewritten with the used language in figure 1.3. The first change, as explained above, is the absence of visibility modifiers in methods declaration. Instead, the programmer defines a maximal class policy, that reflects the most permissive privilege a user can have. The second change is the creation of authorizations to access the file. Its owner (who created the object) can generate various authorizations with different policies with the new primitive *authorization* and doing so he is granting rights to access the file to whoever can apply those authorizations. The third change is the usage of the new primitive *authorize* to

change the policy of the user view f for the expected one. The primitive *authorize* is a run-time verification that checks if a given authorization validates the intended policy for the file. These are the only points where a run-time check is performed during the entire program execution. The type system statically verifies if the methods are being called according to the current user view policy.

```
class File : { (open; (read + write)* ; close)* } {
    void open() {...}
    void read() {...}
    void write() {...}
    void close() {...}
}

void create(){
    File f = new File;
    Auth readOnly =
              authorization(f, {open; read*; close});
    Auth writeOnly =
              authorization(f, {open; write*; close});
    DoRead(f,readOnly);
    DoWrite(f,writeOnly);
}

void DoRead(File f, Auth a){
  authorize f:a
     case {open; read*; close} :
        { f.open();  f.read(); f.read(); f.close() }
      case error: { }
}

void DoWrite(File f, Auth a){
 authorize f:a
     case {open; write*; close} :
        { f.open();  f.write(); f.write(); f.close() }
      case error: { }
}
```

Figure 1.3 Rewritten File class declaration and usage with our solution.

As we can see, we now can verify if methods *DoRead* and *DoWrite* use the file as the intended protocol. Not only that, by applying authorizations with the *authorize* primitive we

also know that only authorized users are accessing the protected file. If a method was called outside an *authorize* case block by a non-owner user, it would be considered an unauthorized method call and captured by the typechecker as a typing error.

In chapter 5 we describe a possible solution to how authorizations could be treated by a runtime environment in order to keep them safe and trustworthy. We also discuss how user views could be implemented to ensure the same security properties provided by the type system on an environment where we can not trust the compiled code. Finally we present the implementation of the typechecker for our type system.

1.3 Contributions

This work's contributions are as follows:

- The main contribution of this work is the conceptualization and formalization of a novel type system for access control to objects in an object oriented language (section 4.4). Although we developed it for a customized language, it can be adapted to traditional object-oriented language, since we rely on an implicit type and effect system, with small changes to the original language grammar.

- Furthermore, we introduce user views as a link between an object and the user privileges to that object. References are therefore protected with the additional information about privileges and objects can only be accessed according to the corresponding privileges. Additionally, we introduce authorizations to allow dynamic modifications of user privileges within user views.

- The type system developed in this work is able to verify that all protected objects are used according to the corresponding privileges. To this end, we formulate this type system as a type and effect system, where types are extended with the corresponding object privilege. As such, the contribution is a study of a type and effect system for an object-oriented language capable of performing the objectives mentioned previously.

- We present a soundness proof for this type system. Although following an usual methodology for proving soundness, based on operational semantics, the proof is necessary to assert the correctness and safety of a program as subject reduction property.

- We also developed a prototype typechecker for the language to automatically verify programs. Thus, this typechecker is a tool to verify access control to objects in our object-oriented language.

2. Context and Related Work

This thesis covers two areas: security and type systems. In this chapter we analyze both areas to extract important concepts and underline some of the research done in these topics.

We want to create a type system to guarantee access control to objects. So, we need to understand what is the role of access control on a security system and how type systems are used to ensure security properties.

2.1 Security

In this section we will give an overview of security in computer software. We describe the properties of a security system and detail a common abstract technique to ensure them - access control. Then, we categorize how security verifications can be implemented into two groups, dynamic and static verifications, to highlight the advantages and limitations of type systems, as well as to understand the alternative approaches. Finally, we revised static verifications, namely language-based security, since a type system is categorized as a language-based technique to ensure security.

2.1.1 Security Properties

In the area of computer security there are several concepts that define the various elements of a security system. Any information or resource available by the system is called *data*, which can be protected. The elements that try to access the data are classified as *subjects*.

A security system has properties that define it. It is based on those properties that the system is evaluated. There are three properties that all security systems have and must ensure: confidentiality, integrity and availability (CIA Triad)[16].

Confidentiality is the assurance that protected data is accessible only to authorized subjects. This is necessary to ensure that no protected information or resource is disclosed to unauthorized subjects. Examples of mechanisms to apply confidentiality are access control, preventing unauthorized subjects to access the data, and cryptography, protecting the data with encryption so that only subjects with the corresponding key can access it.

Integrity is the assurance that protected data is maintained authentic and correct. This property prevents erroneous changes to protected data by both unauthorized and authorized subjects trying to perform unauthorized actions. Users of the protected system can be sure that the data obtained from the system is safe and reliable, because no unauthorized modifications were made. Example of mechanisms to apply integrity are access control, preventing unauthorized subjects to access and modify the data, and cryptography, protecting the data with encryption to confirm that the data was not modify during transit.

Availability is the guarantee that protected data is available to authenticated subjects when

requested. This property is often underrated for compliance of the previous properties. However, it is important that resources remain available to those who can access them, since otherwise the objective of the protected system is not fulfilled. An example of a mechanism to enforce availability is data replication.

Based on these properties, other concepts emerge. *Identification* is a token of a subject identity that will serve to authenticate the subject before the system with an *authentication* process. Additionally, there is an *authorization* process, which approves an action for an authorized subject. These two processes characterize a security *access control* system.

2.1.2 Access Control

To meet the confidentiality and integrity properties of a security system, it must ensure that the protected data is handled only by authorized subjects [16]. Without these verifications, the data would be unstable, incorrect and unreliable and could lead to unpredictable execution errors and behavior. So, an access control system [40, section 1.2] must be capable to authenticate subjects, rejecting unauthorized ones, and verify if they have the authorization to execute the required operations to access the protected data. Therefore, there are two processes that the system must have to accomplish its goals: Authentication and Authorization.

Authentication [16] is a process to verify the identity of a subject. If the subject is known to the system, then he can perform operations. Otherwise, his access to the system is denied. To achieve this, the subject needs to have a witness identity which identifies him uniquely. This token can take several forms, depending on the system in question. Interactive applications typically rely on user name and password to identify a subject. In a communication protocol, the manipulation of encrypted messages with private or shared keys enables the authentication of participants.

Authorization [16] is a process to check if an authenticated subject is able to perform the requested operation. This operation is performed only if the subject has the relevant permissions. Thus, the system must have a mechanism to manage permissions, assigning them to authenticated subjects. Typically, these associations follow the Principle of Least Privilege that says that each subject must have the minimum permissions possible to carry out their operations.

There are three methods of implementing the authorization process which define the type of access control [47, 22, 40]: discretionary access control, mandatory access control, and role-based access control. An access control system implementation can follow any combination of these approaches, although with each approach the system becomes more complex.

In a *Discretionary Access Control* (*DAC*) [47, 40] the access permissions to protected data are controlled by its owner. In a file system, for example, a file owner can define who has read, write and execute rights over that file. Although heavily used, this method has two major disadvantages. First, permissions to read are transitive. A subject with read permission can copy the protected file data to another file owned by that subject and then give another subject permission to read over that file without knowledge of the original owner. Secondly, this method is vulnerable to "trojan horse" attacks. A user program generally has the same rights as that user

and so such programs can perform malicious operations over a file owned by that user. This method implementation follows two approaches mentioned before: access control lists, where subjects are associated to their rights, and access control based on tokens or certificates emitted by a certifying entity and given to subjects as proof of authorization to access the protected data.

In a *Mandatory Access Control* (*MAC*) [47, 40] the focus is on the data classification. The system assigns labels to resources, creating permission levels. Additionally, the subjects are assigned to labels, for which access is controlled whether a subject has the same level or higher than the resource he is trying to access. The levels of secrecy used by companies and authorities are examples of application of this method. The famous "Top Secret" defines information that is very restricted and can only the accessed by subjects with the appropriate security level. The main difference between this and the previous approach is that permissions are assigned by the system (or administrator) and not by the data owners. This leads to limited control over resources, because only the system can manage the permissions.

In a *Roled-based Access Control* (*RBAC*) [22] permissions are also managed by the system. However, the difference is the expressiveness of these permissions. While in MAC they are simple privileges to read or write based on labels or levels of access, in RBAC permissions are assigned to roles, which may reflect complete transactions. These roles are assigned to subjects and they may have more than one role. Thus, the subject can only perform an operation if it exists in one of its roles. This approach makes it more flexible management of permissions, because one can assign existing roles to new users, being widely adopted and implemented in many systems.

Analyzing the approaches described here, our solution has characteristics of an access control based on tokens (DAC) and a role-based access control. In the first case, the similarity between tokens and our authorizations is the fact that authorizations are first class values and can be treated as checkable tokens to perform operations. However, in discretionary access control security policies can only be granted by the owner of the protected data and are too simple (read or write policies). On the other hand, in role-based access control, policies are more sophisticated and may represent complete usage protocols, as our authorizations. Additionally, the concept of role can be obtained in our solution if the encryption technique used to generate authorizations use asymmetric keys, where one of the keys is considered a role. Only processes that know the key can apply the authorization to gain access to the data. The difference between the approaches is the usability of roles and authorizations. Roles are fixed policies, managed by the system; authorizations are first class values, managed by those who have them.

There is several research done on access control. Some of them are outside of this thesis' scope, but they are worth mentioning. They are all based on ensuring access control and authentication with logic. Abadi et al. [4, 38, 5, 7] use logical expressions with the form *P says E*, that is, the principal *P* makes expression *E* true, or, in terms of security, principal *P* authorizes *E*. Then, they construct *E* to reflect access control expressions, such as, *MayAccess(alice, file)*. So, the expression *bob says MayAccess(alice, file)* states that bob authorizes alice to access a file. These concepts inspired other works, such as [34, 18, 17, 25, 35]. As we are more interested in analyzing type systems, we will focus our overview on [25, 35] in section 2.2.

2.1.3 Approaches for Guaranteeing Security

So far we covered abstract notions of security. Now we will describe what and where verifications can be performed to ensure not only access control, but all security properties.

There are three levels of abstraction to where we can verify security. We could create system models using various processes calculus and verify the desired security properties using model checkers and then implement the system according to those models; use static analysis to verify if the code follows the desired properties; or develop runtime security systems that monitor the protected data. These techniques can be categorized in two groups: dynamic verifications and static verifications.

Dynamic verifications encompass all verifications done at run-time and are based on reference monitors that control the access to data. A common method for doing so is Access Control Lists (ACL) [40, section 1.2.5], associating a list or table to resources where the corresponding rights for different users are stored. In a Unix file system, for example, each file has a permission list for groups of users. When a user wants to read a file, the security system checks if the user has the right to do so in the file ACL and, if it fails, the access is denied. On another example, a router has an ACL associating the host ports to participants addresses. So, it is possible to verify the access privileges for a given address to a port. Another dynamic method used to implement security is the use of security tokens (authorizations or certificates). A token can take any form and its goal is to prove that a user has the permission to access a protected data. As such, resources themselves have no information about existing tokens but the system must have a trust circle including the entity that creates those tokens. Once checked the authenticity of a token, the subject has access to the resource without the need for more checks.

The static verifications on the other hand, are based on logical rules imposed on programs structure and operations before execution. System models can be built based on the system architecture using, for example, process calculi [31, 41, 30, 42, 27]. Since the system is not actually running, model checking is considered a static verification technique. It is used to verify the correctness of computer systems and ensure that some security properties hold on every possible execution of a system. However, the actual system implementation can sometimes not be in agreement with the model and have errors outside its domain. To address these difficulties, code verification techniques are used with appropriate tools based on restrictions and conditions imposed by programmers and analysts [46, 39, 10]. The notions of pre and post conditions reflect the cause and consequence of each instruction or method and are the foundation of these tools. On the other end, compilers can be extended to not only transform the code to another language but also to verify if the code follows pre-established rules that reflect security concerns. A type system [15, 48, 49] checks if a program source follows typing rules, preventing type errors during execution. These rules can be complex enough to control and verify an object behavior on an object-oriented language. Indicating methods visibility is a simple language based access control; a private method can only be called within the class it is defined.

All these techniques can be used when developing a system in order to obtain the maximum security. Before implementing the system one can design a simplified model for it and use

model checking to guarantee correctness and some security properties. During implementation, static analysis of code can be perform in order to guarantee that the code is safe and prevent some execution errors. Finally, during execution we can use reference monitors to guarantee security properties impossible or very hard to verify during the previous steps.

2.1.4 Language-based Security

We focus our work on a type system that provides access control to objects in an object-oriented language. As we have seen, this is a static verification technique to guarantee security. Many other static techniques to enforce security properties exist based on programming languages. We now see why these language-based security approaches are considered useful [36, 50].

Traditionally, computer security has been enforced by operating systems but, as they grow in complexity, it becomes increasingly difficult to handle security at this level. Consequently, modern attacks often succeed in fooling operating-system security mechanisms by performing application-level attacks. To defend against this kind of attack one must apply application-level security or language-based security.

Language-based security [36, 50] is based on program rewriting, program analysis and certifying compilers.

In-lined execution monitors [20, 19, 21] takes rewriting as its base concept. It merges the application code with checking code to ensure the desired security requirements, creating a new secured application, which is guaranteed not to violate those requirements.

Program analysis [46], such as typechecking and dataflow analysis, can be used to reason statically about run-time behavior of code, eliminating some run-time security checks. Dataflow analysis [29] reasons about run-time values of expressions at different program points during compile-type by representing the program as a control flow graph, where nodes are program statements and edges are flow of control. The information obtained from the graph is then used to analyze various system properties and detect code anomalies, such as null pointers detection (pointer analysis), and detecting uninitialized variables. Type systems [15, 48] prevent the occurrence of errors during a program execution by forcing the programmer to write the program in conformance with typing rules. A typed variable can only be associated with values of the same type. Typing rules can also impose security constraints that need to be met in order to run the program safely. Since this work is based on a type system, we further describe type systems in section 2.2.

Proof carrying code [43, 44] is a technique used to safely execute untrusted code. The code receiver establishes a set of safety rules that guarantee safe behavior of programs, and the code producer supplies a formal safety proof that proves, for the untrusted code, the compliance to the safety rules. If the proof is valid, the untrusted code is safe to execute. A certifying compiler [32] is a compiler that, given a source code satisfying a particular safety rules, produces the program code and a certificate that contains the safety proof.

With these techniques, language-based security is a capable mechanism to enforce a broad class of fine-grained security policies using both high-level and low-level languages.

2.2 Type Systems

The main goal of a type system is to prevent the occurrence of errors during a program execution. As such, one step to achieve this goal is the definition of an execution error [15].

When the main goal is achieved, i.e., a program will not contain execution errors due to its typing, we can say that the language is *type sound*. When properly developed, a type system becomes a tool to formally verify the implementation of important properties for a programming language.

A variable type defines the limitations and restrictions of values that the variable can hold during a program execution [15, 48]. For example, a variable x of type *Boolean* can only hold boolean values during all program execution. Fictionally, expressions that use these values such as *not(x)* have semantic meaning that will be preserved throughout the program execution. Programming languages where one can assign types to variables are classified as *typed languages*.

In this section we examine the concepts and properties of type systems. First, we need to understand what execution errors are. Not all execution errors can be captured with a type system. Then, we analyze the properties offered by type systems to understand what we can expect from them. Finally we need to know how to formalize a type system and prove its correctness.

This overview is based on traditional type systems where judgements take the form of $\Delta \vdash e : T$, meaning that expression e has type T under the typing environment Δ. When more sophisticated type systems are developed this simple form usually changes to custom ones. The work of this thesis is based on type and effect systems, where additional information is added to types to allow the capture of more errors. Thus, we also describe some concepts of type and effect systems.

2.2.1 Execution Errors

As we said, the objective of type systems is to prevent execution errors. So, the definition of execution errors is one of the most important steps when developing a type system. In spite of this, we need to understand what kind of errors a type system is able to prevent.

Execution errors are usually distinguished in two classes [15]: *trapped errors*, the kind of errors that cause the interruption of a program, and *untrapped errors*, the kind of errors that go unnoticed and later cause bad execution behavior or wrong results. Examples of trapped errors are division by zero and the access to an illegal address. In these cases, the computation typically stops immediately. Examples of untrapped errors are accessing data past the end of an array in absence of run-time bounds checks and jumping to a wrong address that does not represent an instruction stream.

If a program fragment does not cause untrapped errors to occur it is considered *safe*. Languages where all program fragments are safe are designated *safe languages*. Typed languages may enforce safety by statically excluding all programs that are possibly unsafe or use run-time checks. Besides capturing untrapped errors, typed languages can capture many trapped errors.

Due to this property, we may designate a subset of the possible execution errors as *forbidden errors* that include all untrapped errors plus some trapped errors. A program fragment is said to be well behaved if it does not cause any forbidden errors to occur, and ill behaved otherwise. So, a well behaved program is safe.

The process to check if a program implemented with a typed language is well behaved is called *typechecking* and the algorithm that performs this checking is called the *typechecker*. Only when a typechecker succeeds typing a program we can be sure that a program is safe to run [15, 48].

2.2.2 Type Systems Properties

Annotations about program behavior can range from informal code comments to formal specifications subject to formal proof. Types are positioned in the middle of these extremes. They are more accurate than comments and more easily verified that specifications. The basic properties expected of any type system are [15]:

- A type system should be decidably verifiable, i.e., there must exists an algorithm able to ensure that a program is well behaved. Errors should be detected before program execution.
- A type system should be transparent, i.e., the result of a typecheck should be predictable by the programmer. If there is a typing error, its cause should be obvious.
- Type systems should be enforceable, i.e., type declarations should be statically checked as much as possible, and otherwise dynamically checked.

2.2.3 Formalization and Soundness Theorem

To guarantee that typed programs are really well behaved, a type system must be formalized with typing rules. Once done, we can prove a type soundness theorem that says that well-typed programs are well behaved. If the theorem holds, the type system is considered sound [15, 48].

The type system formalization and the proof for the said theorem are achieved formalizing the whole language with four components:

- The Language Syntax to establish how a program can be constructed;
- Operational Semantics to determine the language constructs computations;
- Typing Rules to define how each language expression is typed;
- Proof of Type Soundness to ensure that well-typed programs are well behaved.

The operational semantics and typing rules have a strong relationship: each expression computation and type must be related. This relationship is the base for the type soundness proof [56, 15]. Faulty expressions follow from the cases where the operational semantics cannot be applied. Type systems should be able to capture these errors during compile-time, preventing them from occurring during run-time.

In this thesis we follow the standard technique to prove type soundness [56]. Each program intermediate evaluation state is by itself a program, result of a small step reduction of the original program. A reduction may not terminate ($e \Uparrow$) or it can end in a state where no reduction is possible ($e_1 \twoheadrightarrow e_2$ ou $e \not\twoheadrightarrow$). This final state represents the evaluation result or a typing error. Thus, proving the type soundness theorem is verifying that well-typed programs only evaluate to well-typed program when the reduction reach the final state. An expression that contains a typing error is called a *faulty expression*.

Thus, the theorem proof is based on two fundamental theorems about the type of states. The first is to assert that every intermediate state has the same type as the previous state, i.e., reduction preserves typing. This theorem is called *subject reduction* [56]:

if $\vdash e_1 : T$ *and* $e_1 \rightarrow e_2$ *, then* $\vdash e_2 : T$.

The second theorem is to assert that the computation of closed well-typed expressions will never get stuck, although the possibility of not terminating still exists, i.e., either the expression reduces or it is a value. This theorem is called *progress*:

if $\vdash e_1 : T$, then either $e_1 \rightarrow e_2$ or $\vdash e_1$ is a value.

With these two theorems, we can conclude that if a program is well-typed and terminates, its result is a value that will also be well-typed with the same type. Therefore, the type soundness theorem is based in the syntactic connection between the result of a reduction and its type.

As a result of the theorem, by progress we can observe that faulty expressions are not typed and so the subject reduction theorem cannot be applied. Hence, a well-typed program does not contain typing errors.

In summary, to prove type soundness it is necessary to perform the following steps:

- Define error expressions;
- Prove subject reduction theorem;
- Prove progress theorem.

2.2.4 Type and Effect Systems

Conventional type systems associate types to variables, constraining the values that those variables can possess. They provide a simple yet effective methodology of specifying program properties. However, many properties remain unchecked with this kind of type systems.

Type and effect systems [26, 55, 45] enhance the type information with annotations to express more system properties. In these type systems, typing judgements associate a program with a type, limiting the range of values that the program can compute, and an effect, which describes the side-effects that the program may have. The concept was first introduced by Gifford and Lucassen in [26] as an effect system, where each expression had an effect class in addition to its type that described how its value was computed. The effect system was used to analyze the side-effects of each expression. For example, if an expression was typed as an OBSERVER,

it could observe side-effects, but it could not cause them. As such, the effect system allowed tracking the manipulation of dynamically allocated memory.

Over the years, the notion of type and effect systems has evolved [55, 45] and various projects develop them for different purposes. Skalka and Smith [52] combine a type and effect system with model-checking techniques, where the type system infers a sequence of events - to what they called history - and then use model-checking to verify the desired properties. A similar work was done by Bartoletti et al. [11], where the extracted program history is verified against access control policies, and then extended by Bartoletti et al. [12], where the history can handle fresh names, enabling a fine resource usage analysis. Flanagan and Freund [23] and Abadi et al. [6] use type and effect systems to enforce a locking discipline to prevent race conditions in Java. In [23], the typing rules track the set of locks held at each program point, and typing an expression could add or remove locks from that set. In [6], expressions are typed with effect P or U, where P means that the evaluation of e accesses only protected locations and the evaluation of U accesses only unprotected locations, introducing a distinction between "P code" and "U code". Then, the effect of typing an expression could be changing from P to U or vice-versa.

As we can see, type and effect systems can be used to ensure a variety of safety and security properties. In this dissertation, we design a type and effect system to guarantee that authorized individuals use protected objects according to their privileges, allowing statically verified access control.

2.2.5 Type Systems for Security

Type systems have been developed to reason statically about system properties and requirements for both modeling and programming languages. We will now make an overview of type systems used to ensure security properties. We first enter the world of modeling languages, or process calculi [31, 41, 30, 42, 27, 17].

Abadi [3] extends the spi-calculus [27] with a type system that reasons about the confidentiality of values. This type system considers three types for values: *Public*, to represent data that can be transmitted to any process; *Secret*, to represent data that cannot be disclosed; and *Any*, that represent arbitrary data. So, the type system verifies that only values of type *Public* are transmitted on public channels. For a value of type *Secret* to be transmitted on a public channel, it must be part of an encrypted message together with a value of type *Any* and a value of type *Public*. The safety theorem states that if a process is well-typed on an environment E that contains only values with types *Public* and *Any*, and if δ_1 and δ_2 are two substitutions of values for the variables in E, then the processes $P\delta_1$ and $P\delta_2$ are testing equivalent, i.e., an attacker could not distinguish the two processes, thus ensuring confidentiality.

Bugliesi et al. [14] use a group calculus (πG), that allows the definition of process groups. The new type for this calculus has the form $G[T\|\Delta]$, where G identifies the group that hold the control of values of that type, T defines the structure of those values, and Δ represents a delivery policy, stating how data of that type transit on system channels. As an example, the type

`j : Job[filedesc‖ Spool -> Print -> Client ]`construes j as a file descriptor to be first delivered to the spooler, then passed on to the printer, and only then re-transmitted back to clients for notification. Additionally, channel types have an annotation to describe if it is an output, input or mixed channel. Following the example, the spooler would have a channel of type `s: Spool[ (Job[filedesc ‖ Print -> Client])`$^{\texttt{rw}}$ `] ‖ - ]`. As such, given the two type assumptions for j and s, the type system guarantees that transmitting j over s is a well-defined, and legal, operation.

Fournet et al. [25] extend the spi-calculus with constructions using the logic language Datalog, allowing the definition of inert processes designated by *statements* and *expectations*, that define the system's authorization policy. The following example is an expectation that reflects the privilege of x to read y:

```
expect CanRead(bob,handbook)
```

The methodology consists on creating statements as privileges declaration, usually following a dynamic verification of credentials, and expectations as privileges checks to continue the execution to, for example, access a protected resource. The previous example could be used as following:

```
CanRead(bob,handbook) | out c(bob,ok)
| in c(x,y); expect CanRead(x,handbook)
```

The goal of the type system is to verify if, for each possible execution, all expectations are logically entailed by the set of active statements. To transmit statements between processes, a special value *ok* was created as every message component, without any semantic meaning. Its type is $Ok(S)$, that verifies if it is safe to assume the logic expression S. Then, the type for the channel c from the example would be:

```
Ch((x:Un, Ok(CanRead(x,handbook))))
```

As such, the main result is the insurance that well-typed processes follow every internal policy defined by the logical expressions.

Now we will step into the world of programming languages, where type systems are broadly used.

Skalka and Smith [51] developed a language based on the λ-calculus with concepts of the Java Security Architecture. They include the constructions *letpriv* to (statically) add privileges to the environment and *checkpriv* to check if a privilege exists. So, expressions that have *checkpriv* as sub-expressions are typed with $\tau_1 \xrightarrow{\Pi} \tau_2$, that is, they are functions with a parameter of type τ_1, return type τ_2 and need the set of available privileges Π to execute. To that end, the authors add a new environment to the typing judgement to keep track of available privileges. The typing of the construction *letpriv* π *in exp* adds the permission π to that environment, followed by the typing of expression *exp*. The typing of the construction *checkpriv* π *forexp* only

succeeds if the privilege π is on the environment. A similar work is done by Banerjee and Naumann [9, 8], where the privileges are logic expressions, defined by the programmer. A read privilege, for example, could be defined by a simple *CanRead* flag.

Swamy et al. created Fable [54], a programming language in which programmers may specify security policies and reason that these policies are properly enforced using a type system. Security labels are associated with the data, representing the security policy specifically for that data, and are defined and manipulated only on a separate part of the program called the *enforcement policy*, allowing the creation of complex labels. As an example, the programmer could define the label *ACL(Alice,Bob)*, reflecting an access control list, and associating it with an integer variable, resulting in the type *int{ACL(Alice,Bob)}*. This type says that the variable can only be accessed by *Alice* and *Bob*. To interpret the labels, the programmer must describe their semantics, in the form of a policy. The given example would have the following policy:

```
policy access simple (acl:lab, x:int{acl}) =
    if (member user acl) then {o}x else -1
```

This function receives a label like *ACL(Alice, Bob)* and an integer protected with that label. If the current user belongs to the list represented by the label then the value of the integer is returned without the label, that is, the access was granted. The typing rules are divided into two contexts: the application (green) and policy (blue). Some language constructions can only be typed on a specific context.

Bengtson et al. [13] define a calculus, and the corresponding adaptation to the programming language F#, with refinement types, whose notations are similar to the Fable's labels. Types are refined with logical expressions that can also be defined by the programmer. As an example, a refinement that reflects the privilege to read or to write a file could be defined as follows:

```
type facts =
     | CanRead of string
     | CanWrite of string
```

So, *CanWrite("name.txt")* represents the privilege to write over the file named *"name.txt"*. These facts are used to refine the base types, for example:

```
val read: file:string{CanRead(file)} -> string
val delete: file:string{CanWrite(file)} -> unit
```

So, the function *delete("name.txt")* can only be invoked in contexts where the fact *CanWrite ("name.txt")* was previously established. Additionally, there are commands to verify if a restriction is true (*assert*) and to assume that a restriction is true (*assume*). With the later, the programmer can define logic implications between facts, as the following example:

```
assume ∀ x. CanWrite(x) ⇒ CanRead(x)
```

The key point of the type system is the conformance of types of function's parameters, namely checking if the facts of a refined type were previously presented, using first order logic.

Jia et al. [35] developed the AURA language with a type system that contains a logic to create authorizations, similar to the one created by Abadi [7]. Its goal is to prove that a subject can access a protected data. To that end, they use depended types that allow using language values on policies. For example, the function *playFor* plays a music *s* on behalf of a subject *p* and is typed with the following type:

```
(s :Song) -> (p: prin) -> pf (self says MayPlay p s) -> Unit
```

Authorizations can be created using the construction *sign(a, P)*, where *p* is a principal and *P* is a policy, resulting on the authorization *a says p* with the same format as its type, *a says p*. Hence, some typing rules have premises to check if the type of a sub-expression is *a says p*.

These works were the main source of motivation and inspiration in the development of this dissertation. We now summarize their similarities with our work and some inspirations.

- The work done in [3] served as a general motivation for developing a type system for security. It shows that type systems are capable of ensuring such properties.
- In [14] the authors also use a kind of policy based on states, though their represent channels where a resource must transit in, while ours represent what methods are available to be invoked.
- In [25] the authors use statements and expectations, with similar behavior as our primitives authorization and authorize. Statements and authorizations both represent a token that expectations and authorize primitives use to grant access to a resource.
- In [51] a function is typed with $\tau_1 \xrightarrow{\Pi} \tau_2$, stating that it needs the privileges Π to execute. Our method call restriction has a similar notation: $\pi \xrightarrow{m} \pi'$, although it is not bound to the method type.
- In [54] it is possible to create labels to protect data and attached them to individual value types. So, each variable has a label associated with it that restricts its usage, much like our policies can control methods calls. However, we use dynamic authorizations capable of changing policies, whereas they define a function to change the labels of values.
- Our notation for user types ($O\{\pi\}$) was inspired by the one used by [13] to refine a type with logical expressions, though these expressions are treated differently from our policies.
- Finally [35] also creates authorizations as values of the language, though they are context dependent and, consequently, are not first class values. Our authorizations, on the other hand, can be stored on data structures, used as an argument to a method call or as a return value.

The main novelty of our approach is the conceptualization of user view: values that hold both object and policy for that object, decoupling the usual explicit access control policies from classes and thus allowing the dynamic modification of policies via authorizations.

3. A Type System for Access Control to Objects

In this chapter we fully describe the developed solution. We introduce the definitions for policy, user view and authorization, giving examples of how they can be used. In the next chapter we will present the technical development of our solution, leading to the type soundness result. We also discuss details and solutions for some of the challenges that arose during this work's development. We finalize by giving an example where this technique is applied.

3.1 Description and Goals

As we have seen in chapter 2, access control is a common technique to ensure security in a system. If a system is implemented using an object oriented language, its resources are designed as objects. So, protecting these objects from malicious usage is a simple way to ensure global system security.

We consider an object usage to be its method calls. We could also consider field manipulation to be part of the object usage, but we assume that these operations are encoded with methods (getters and setters). As such, controlling the access to objects can be done by creating *privileges* that reflect object usage. To specify an object privileges, we extend its type information with implicit notation. This extra information is defined as a *policy*. Furthermore, we can statically verify that an object is used according to the current policy, preventing unauthorized (or unexpected) method calls. References to objects carry more information - they hold not only the location of the object but also a policy. We designate these extended references as *user views*. To access an object, one must do so through a user view and the access is protected with the corresponding policy.

We want the user views to be flexible, i.e., we want to be able to change an object policy dynamically, reflecting the changes of user privileges. To do so, we introduce *authorizations*. The manipulation of authorizations is a programmer responsibility, though the type system aids this task by statically verifying if authorizations can be created in a given environment.

3.2 Policies

Our definition of policy is as follows:

Definition 3.2.1. ***Policy.*** *A specification of the privileges to access an object, describing which methods may be invoked at a given moment.*

Each class will have a *maximal class policy* declared by the programmer to indicate the most permissive privilege allowed for instance objects of that class (except the self reference *this*). A new object will have associated with it a maximal class policy, since the owner of the object has full access to it.

To describe a policy, the programmer can use any policy language as long as this language defines three properties necessary for the developed type system. As such, the expressiveness of a policy only depends on the language used to define it.

In figure 1.3 we used regular expressions to describe the policies:

```
{open; write*; close}
```

The properties that a language must define are the following:

1. Validation. This simple property states that given a set of methods, a policy must be valid with that set, i.e., an object policy contains only methods of that object. We write this property as follows:

   Let M be the given set of methods and π a policy, $M \vdash \pi$.

2. Subset/subtype/simulation. This property is necessary to create new authorizations. It states that given two policies (π and π') it must be possible to compare them in order to verify if one (π') performs every action of the other (π). Depending on the language used, it reflects, for example, the notion of subtype or subset. We write this property as follows:

   Let π and π' be policies, $\pi <: \pi'$.

3. Progress/step/reduction. Each time an object method is called, its policy can change. This property reflects that possibility. It states that given a policy and a method's name, the policy must be able evolve into another policy (or the same) using that name. We write this property as follows:

   Let π and π' be policies and m a method's name, $\pi \xrightarrow{m} \pi'$.

To exemplify the expressiveness of different languages, we present three languages that can be used with this type system.

3.2.1 Set of Methods

The first example language is Set of Methods and it can be seen as a set of the traditionally public methods. A policy in this language is a set of available methods:

```
{open, read, write, close}
```

Though not very expressive, it gives simple control over which methods can be called. Additionally, the set does not change when a method is called, reflecting unlimited access to the methods.

The properties needed by the type system are defined as follows:

1. Validation. To verify the condition, the policy set must be a subset of the given method set.

   Let M be the given set of methods and π a policy set, $M \vdash \pi \triangleq \forall\, m \in \pi \;\Rightarrow m \in M$.

2. Subset. Similar to above but between policy sets. The traditional subset definition.

   Let π and π' be policy sets, $\pi <: \pi' \triangleq \forall\, m \in \pi \;\Rightarrow m \in \pi'$.

3. Progress. The set does not change when any method is called.

   Let P be the policy and m a method being called, $P \xrightarrow{m} P$.

As all properties are defined, this policy language can be applied with the type system to create safe programs where every method call is statically checked if it can be done according to the current object policy.

This language is not very powerful. It is similar to the traditional approach of modifiers, though the dynamic property of policies makes it more powerful than the former. To make it more complex, we add extra information and create another language: Method Call Count.

3.2.2 Method Call Count

The previous language is not very expressive. As such, we introduce another language that has an extra layer of expressiveness: the ability to count method calls. The objective is to specify how many calls can be done for a method and statically verify if there is not more calls that the count. A policy is again a set of methods but with extra information for the count:

```
{open 1, read 2, write 2, close 1}
```

In this example, the *read* method can be called no more than two times when this policy is applied. This is a kind of behavior not possible to describe with traditional type systems. It could be simulated with runtime checks, but that would add overhead to the program, with no guarantee of correctness.

A method call now has an affect on the policy. When a method is called, its count is reduced by one. If it reaches zero, the method is removed from the set.

As before, the properties needed by the type system are defined as follows:

1. Validation. To verify the condition, method from the policy set must be elements of the given method set.

   Let M be the given set of methods and π a policy set, $M \vdash \pi \triangleq \forall\, m \in \pi \;\Rightarrow m \in M$.

2. Subset. In this language, a policy π is subset of another policy π' if a) every method of π is an method of π' and b) for every method m in both policies, the count of m in π is equal or less than the count of m in π'.

   Let π and π' be policy sets, $\pi <: \pi' \triangleq \forall\, m\;\; c \in \pi \;\Rightarrow m\;\; c' \in \pi' \wedge c <= c'$.

3. Progress. A method call reduces the method count in the policy.

   Let π be the policy and m a method being called,

   $\pi = \{M_l,\ m\ \ 1,\ M_r\} \xrightarrow{m} \pi' = \{M_l,\ M_r\}$;

   $\pi = \{M_l,\ m\ \ c,\ M_r\} \xrightarrow{m} \pi' = \{M_l,\ m\ \ c-1,\ M_r\}$.

This language defines all needed properties and therefore can be used with the type system. The result is the ability to statically verify if object methods are being called at most n times when a policy is enforced.

Though this language enables additionally expressiveness to the language, we still cannot specify the order of which we want to call methods of an object. To do so, we need another policy language that uses regular expressions.

3.2.3 Regular Expressions

The initial language used to develop this type system was regular expressions, where terms are methods. Its powerful semantics allows the creation of complex policies that describe a behavior of an object. Not only it shows which methods can be used, but also when they must be called, creating dependencies among methods:

```
{open; write*; close}
```

In this example, when this policy is enforced the only available method to be called is *open*. Only after it is called the *write* and *close* methods can be called, describing, here, the privilege to only write in a file.

Therefore, a method call has an affect on the policy. When it happens, the policy evolves to the next policy in the sequence where the method is the head. A list of method calls creates a possible policy path.

The properties are defined as follows:

1. Validation. To verify the condition, terms from the policy must be methods of the given method set.

   Let M be the given set of methods and π a policy, $M \vdash \pi \triangleq \forall\, m \in \pi\ \Rightarrow m \in M$.

2. Simulation. We say that a policy π' simulates another policy π if for every possible path of π, π' can perform the same path.

   Let π and π' be policy sets, $\pi <: \pi' \triangleq \forall\, m : \pi \xrightarrow{m} \pi'' \Rightarrow \pi' \xrightarrow{m} \pi'''$ and $\pi'' <: \pi'''$.

3. Step. A method call is a deterministic step in a policy according to the operators semantics.

   Let π be the policy and m a method being called, $\pi \xrightarrow{m} \pi'$.

This language is expressive enough to describe very complex policies. It can describe the previous languages policies but for the second language it would be verbose and extensive to describe the same policies. Nevertheless, it is a powerful and yet simple language to describe policies and so we will use it for every example in this document.

3.3 User Views

We now explain how policies are used in a program. The definition of user view in this thesis context is as follows:

Definition 3.3.1. ***User View.*** *The association between an object and a policy. The only way to access an object.*

We denote a user view as $(l,\ \pi)$, where l is the object location and π the current policy.

In terms of security, a user view can be seen as an authenticated user. Associated with that user is its current privileges to an object. Only authenticated users can access protected data and so objects can only be accessed through user views (see figure 1.2b). User views with empty policy are considered unauthenticated users and thus cannot be used to call methods.

The traditional approach to object access control focus on visibility modifiers, such as Java modifiers [28, section 6.6] public, private, protected and default, to specify where methods can be called and fields accessed. These modifiers are static (figure 1.2a) and are the same for all programs. User views, on the other hand, allow a simpler class declaration, without any modifiers for methods and fields, and allow to modify the objects visibility dynamically using policies. The consequence is that each program views an object differently, according to its current policy.

So, all variables now hold user views instead of directly hold objects. The initial expression of a variable is now extremely important, since it defines the initial policy for the user view. Method parameters are considered unauthenticated access to an object and, as such, the user view will have empty policy as its initial policy. Fields will also hold user views, and their policy handling is one of the big challenges of this work (more in section 3.5).

The following example shows different variable declarations:

```
class A : { f; g* }   // (1)
{
   int f() { 0 }
   int g() { 0 }
   int h() { 0 }
}

class B : { m* }
{
```

```
    A fd =
        let x = new A in  // (2)
           x.f(); x       // (3)

    int m(A par) { 0 } // (4)
}
```

The initial expression of field *fd* is a variable binding (2), which declares the variable x with a new object of type A. As such, x will have the maximal policy class of A, $\{f; g*\}$ (1). The field *fd* will have the same policy as the result of (3), which is $\{g*\}$, since f was called by x. Finally, the parameter of method m in (4) will have an empty policy, since we do not (statically) know its value.

As we can see, although the variables have the same base type A, their policies are different. Hence, the visible methods for each variable depends on their policy state. The parameter *par* has no visible methods and the field *fd* has one visible method, g. Note that method h is not visible by any policy.

3.4 Authorizations

A user view is either created with an empty policy (a method parameter, for example) or with a full access policy (when creating a new object). In order to change a policy we use authorizations.

Authorizations are first class values and can be stored in data structures, be a method argument or be a part of a message to other processes as any other value. They contain an object unique identification and a corresponding policy both encrypted in a single value and once successfully applied to a user view, the corresponding policy changes to the one in the authorization, i.e., the access to the object is granted.

The definition of authorization in this thesis context is as follows:

Definition 3.4.1. ***Authorization.*** *A first class value containing an object unique identification and a corresponding policy. Used to gain access to an object.*

In terms of security, an authorization can be seen as a security token to gain access to a resource. If the user can apply it, meaning he is capable of decrypting it, then the access to the protected object is granted according to the policy established in the authorization. As such, when an authorization is applied, the user view's policy is changed to the one in the authorization. Authorization encryption and how it can be implemented is discussed in section 5.1.

To create and use authorizations we add two primitives to our language. The primitive $\mathit{authorization}(x, \pi)$ creates an authorization to access the object of the user view x with the policy π. This does not change any existing value; the policy within the user view x remains

the same. The only side effect of this primitive is the creation of a new authorization containing the identification of the object and the new policy π. The type system will statically verify if the policy is valid with the object methods (first property of a policy language). We consider that anyone can create an authorization, but the new policy must the a subset of the current user view policy (second property of a policy language), that is, we can only create authorizations with equal or less privileges that our own.

The following example creates an authorization to access the object of the user view f with the policy $\{open; read*; close\}$:

```
File f = new File;
Auth readOnly =
          authorization(f, {open; read*; close});
```

Note that the operation is well-typed since the current policy of the user view f has maximal class policy and therefore the new policy $\{$ *open; read*; close*$\}$ is valid.

Once created, an authorization can be applied to gain access to the protected object. The primitive *authorize x : a case* π *:* $\{e_1\}$ *case error :* $\{e_2\}$ attempts to apply the authorization a to an user view x. The attempt will only succeed if the object of x has the same identification of the object in the authorization and if the desired policy is a subset of the policy in the authorization (second property of a policy language). If it succeeds, the computation continues with expression e_1 and within its scope the user view has the new policy π. If it fails, either because the objects are different, the new policy is not a subset of the one in the authorization or the decryption failed, the computation continues with expression e_2. The entire operation can be seen as a request to gain access an object - it either succeeds and the access is granted or it fails and the access is denied. At the end of the *authorize* block, both branches need to produce the same environment, i.e., the state of all policies must be the same at the end of the primitive.

The following example applies the authorization created in the previous example:

```
void DoRead(File file){
 authorize file : readOnly
   case {open; read*; close} :
     { f.open();  f.read(); f.read(); f.close() }
   case error: { }
}
```

Note that this methods is well-typed whether the operation succeeds or fails. The type system is only able to reason about information presented on the code. Variable values, being dynamic information, cannot be verified by a type system. In this case, an authorization does not have static information of the protected object and the policy, since its type is *Auth*. The type system only verifies if in the case expression the object is used according to the given policy. Therefore, additional verifications needed by the *authorize* primitive (to certify if the authorization can be

applied) are done at run-time. However, once these verifications are complete, the computation can safely continue without any further verifications, since the program has typechecked successfully.

3.5 Challenges

In this section we discuss the main challenges that arise with this kind of approach and present our solutions for them.

3.5.1 Fields

The first challenge we will address is how objects fields are affected by the concept of user views.

As we have seen, variables to objects hold user views that link an object to a policy representing the privileges that the variable has to the object. Fields can also hold user views, being a kind of persistent variables over an object's life. However, since fields can have a policy of their own, those policies must be followed for every possible object usage. This presents a problem. As an example, consider the following classes:

```
class A : { (f ; g)* }
{
   int f() { 0 }
   int g() { 0 }
}

class B : { m* }
{
   A a = new A; // a: A{ (f;g)* }
   int m() { this.a.f() }
}
```

If we had a user view for an object of type B with a policy $\{m;m\}$, we could call method m two times. The calls would be valid, since the policy was being followed, but the policy of field a would be violated, since there would be two calls of method f on a row and the policy of a only allowed one.

We approach this problem with a conservative solution that says that field's policies are invariant, that is, each method must guarantee that fields' policies are the same at the beginning and end of the method. As such, the previous example would not be well-typed. A correct version of class B would be:

```
class B : { m* }
```

```
{
   A a = new A; // a: A{ (f;g)* }
   int m() { this.a.f(); this.a.g() }
}
```

Method m is now well-typed, since the policy of a is reestablished at the end of the method.

This solution also solves an aliasing problem, as we will see shortly. However, it has practical consequences: fields' policies must be cyclic, i.e., they must be able to reach the same state at some point. As such, policy languages must take that into consideration in order to be properly used.

A valid question can be formulated: if fields policies must be cyclic, how can we consider constructors (methods that can only be called once when the object is created)? The answer is quite simple: a field policy is determined by its initial expression. To demonstrate this, lets consider the following class, where the method f is considered the constructor:

```
class C : { f; (g + h)* }
{
   int f() { 0 }
   int g() { 0 }
   int h() { 0 }
}
```

As we can see, the maximal class policy is $\{f;(g+h)*\}$. So, when creating a new variable of this type, we get a user view with a policy $\{f;(g+h)*\}$, which implies that method f must be called first. Hence, to allow fields to be typed with this class, the field's expression must call that method:

```
class D : { ... }
{
   C c = { let x = new C; x.f(); x } // c: C{(g+h)*}
   int m() { this.c.g(); this.c.g() }
}
```

Invariant fields' policies are particular interesting when writing methods that modify a field value. The following example is a classic set method that changes the value of field c in the previous class D:

```
class D : { ... }
{
   C c = { let x = new C; x.f(); x } // c: C{(g+h)*}

   int set (C other) {
```

```
        this.c = other; // c:{} --> ERROR
        0
     }

     int m() { this.c.g(); this.c.g() }
  }
```

As we can see, this method would not be well-typed, since the user view *other* has an empty policy that is passed to *c*, violating the invariant. The correct implementation for this method would be:

```
  void set (C other, Auth a) {
    authorize other : a
    case { (g+h)* } :
        { this.c = other; } // c: C{(g+h)*}  --> OK
    case error: { }  // c: C{ (g+h)* }  --> OK
  }
```

This reflects an intuitive argument: if we have fixed privileges over some data and it is updated to another data, we must have the same privileges over the new data to be able to perform the same actions.

On the other hand, if the initial policy of a field is empty (with term *null*), it is always necessary to apply an authorization in order to call that field's methods:

```
  class D { ... }
  {
    C c = null ;  // c: C{ }

    void set (C other) {
      this.c = other; // c:{ }  --> OK
    }

    int m(Auth a) {
       this.c.g();  //  --> ERROR
       authorize this.c : a
       case { g;g } :  // --> c : C{ g;g }
         { this.c.g(); this.c.g() }  //  --> c : C{ }
       case error : { 0 }   // --> c : C{ }
         // c : { } --> OK
    }
  }
```

Summarizing, fields' policies are determined by their initial expressions and must remain invariant by all methods of the class.

3.5.2 Self Reference *this*

When we first conceptualized our solution, we consider policies to reflect an object **behavior**, i.e., the sequence of steps (method calls) that needed to be done during the *life* of the object; rather than the owned *privileges*, the sequence of steps that we can do while we have *permission*. These concepts are quite different and the main issue is the properties of the self reference *this*. Lets consider the following example:

```
class E : { (f + g)* }
{
   int f() { this.g() }
   int g() { this.h() }
   int h() { 0 }
}
```

If we considered that policies reflected an object **behavior**, the self reference should also follow the current behavior. So the policy $\{f;g\}$ would not be possible to use, since calling the method f would internally call method g and, in turn, call method h, violating the authorized behavior.

We then realized that this was not in conformance with our original idea. The policy $\{f;g\}$ should be usable, since it is a valid policy. So, we formalized the notion of policy to reflect privileges. Hence, the self reference is a special user view that is able to call any class method without changing the current policy (has full access, $\{*\}$). This does not break the desired security, since a policy is the external view that a user has over an object while the self reference is an internal view. A perfect analogy is to consider the self reference as administrator space, where he can do anything safely, while other user views are user spaces, where only authorized actions can occur. Another analogy is the traditional method visibility, where private fields can only be called within the class and are not visible outside of it. In the previous example, the method h can be considered a private method, since there is no possible policy subset of $\{(f+g)*\}$ that can be created containing the method. However, the self reference *this* should be able to access it, since it has full access.

3.5.3 Aliasing

In programming languages, aliasing consists in the situation in which two variables are assign to the same memory location. Thus, modifying the data through one variable implicitly modifies the values of all aliased variables. In out object-oriented language, we consider locations to only hold objects. Problems can arise when one wants to perform analyses over an object state. Having two different variables pointing to the same object could compromise the analyses results.

So, we need to understand if aliasing affects our solution due to the use of policies. Lets consider the following example:

```
class F : { (g ; h)* }
{
   int g() { 0 }
   int h() { 0 }
}

{
   F f1 = new F;
   F f2 = f1;
   f1.g();
   f2.g();
   f2.h();
}
```

Both $f1$ and $f2$ are user views for the same object, but they will have different policies. Our typing rules make sure that binding a user view from one variable to another will transfer the policy from the original user view to the new user view. In the example, when we create variable $f2$ (a new user view) with the user view's information of $f1$, we remove the policy from $f1$ and give it to $f2$. So, the method call $f1.g()$ would be a typing error, since $f1$ does not have privileges over the object. This guarantees that user views will not share policies that have the same origin (*new* or *authorize*). The following example is a classic example of aliasing hard to track, it shows two variables that have user views to the same object and with the same policy:

```
class G : { ... } {

  int w ( F f1, F f2, Auth a ) {
    authorize f1 : a    // (1)
    case { g ; h } : {
       authorize f2 : a   // (2)
       case { g ; h } : {
          f1.g(); f2.g(); f2.h(); f1.h()  // (3)
       }
       case error { 0 }
    }
    case error { 0 }
  }
}

{
  G g = new G;
```

```
  F f = new F;
  Auth a = authorization(f, {g;h});
  g.w(f, f, a);   // (4)
}
```

The aliasing appears in (4) by calling method w with the same variable f. In the method, during run-time, variables $f1$ and $f2$ will have user views to the same objects. With (1) we are authorized to access the object through $f1$, following the policy $\{g;h\}$, and with (2) we are also authorized to access the same object through $f2$, following the same policy $\{g;h\}$. However, they remain separate views and the method calls from (3) are valid.

Aliasing would also be problematic if the fields' policies would not be invariant. In that situation, a method call from one user view could change a field's policy, invalidating a call from other user view to the same object:

```
class H : { (j ; k)* }
{
   F f = new F;  // f: F{ (g;h)* }

   int j() { f.g() }
   int k() { f.h() }
}

class I : { ... } {

  int w ( H h1, H h2, Auth a ) {
    authorize h1 : a    // (1)
    case { j ; k } : {
       authorize h2 : a   // (2)
       case { j ; k } : {
          h2.j();   // (3)
          h2.j();   // (4)
          h2.k();
          h1.k();
       }
       case error { 0 }
    }
    case error { 0 }
  }
}
```

As before, the aliasing appears if user views of $h1$ and $h2$ have the same object. In the case where both (1) and (2) succeed, the method call (3) would be well-typed, implicitly modifying

the policy of field f, but the method call (4) should be a typing error, since it would violate the current policy of f.

In our solution, on the other hand, fields' policies are invariant, allowing the object's methods to be called at any time from anywhere. So, the previous example is well-typed in our type system, since $f1$ and $f2$ privileges do not interfere with each other, even if they are for the same object.

3.6 Complete Example

In this section we show a program as an example of usage of our type system. liberdade a fazer o exemplo.. ha objectos nao protegidos.. metodos com varios parametros.. string

The example consists on a simplified file system that stores protected files. Each file has associated with it an hashtable where the keys are user names and the values are authorizations. As such, for a user to access a file, it must ask the filesystem for its authorization and then apply it to the file.

The file system is also a protected resource, since it has methods that can change the users' authorizations. As such, the filesystem owner, whom we called Admin, creates authorizations for different accesses to the file system. tamem contem uma hashtable de user pass..

When a user wants to access a file, he must first ask the admin for the corresponding authorization to access the file system. Then, after applying the authorization, he can ask the file system for the authorization to access the file. Finally, it can apply this authorization to be able to call the file methods, though he must follow the authorization policy (his privileges over the file).

3.6.1 File and FileRights

For simplicity reasons, the class file is a simple listing of methods, whose code is irrelevant for the example. The maximal class policy reflects the usual usage protocol of files: first we *create* it (the constructor), then we have to *open* it, then we can either *read* or *write* as many times as we want until we *close* it. We can then open it again, following the rest of the protocol.

```
class File :
{create ; (open ; (read + write)* ; close)* }
{
  int create(string name) { ... }
  int open()  { ... }
  String read() { ... }
  int write(string s) { ... }
  int close() { ... }
}
```

To store the users' authorizations, we create another class - *FileRights* - that has the protected file, an hashtable to associate users' names to their authorizations to access that file, and ready-to-use authorizations, created when the file is created, used by the file system to be associated with users. This class is not protected since it will only be accessed by the file system.

```
class FileRights : {*} {
  File file;
  Hashtable rights;

  Auth readOnly;
  Auth writeOnly;
  Auth readWrite;
  Auth noRights;

  int create(string name) {
    file = new File;
    file.create(name);

    readOnly = authorization(file, {open;read*;close} );
    writeOnly = authorization(file, {open;write*;close} );
    readWrite = authorization(file, {open;(read+write)*;close} );
    noRights = authorization(file, {} );

    rights = new Hashtable;  // ( string, Auth )
  }

  File getFile() { file }
  Auth getUserAuth(string user) { rights.get( user ) }
  int setUserRight(string user, Auth auth) {
    rights.set(user, auth); 0 }

  Auth getR() { readOnly }
  Auth getW() { writeOnly }
  Auth getRW() { readWrite }
  Auth getN() { noRights }
}
```

3.6.2 FileSystem

The FileSystem class is a protected class that has an hashtable associating file names to FileRights objects. The maximal class privilege says that all methods are accessible at any time. A file system serves as a mediator between user and file privileges management, hiding the usage of the class FileRights from users.

For brevity reasons, we only show how creating and accessing a file could be implemented, as well as the management of authorizations for users.

```
class FileSystem
{ (addFile + getFile + getAuth +
   addUser_R + addUser_W + addUser_RW + addUser_N)* }
{

  Hashtable files = new Hashtable; // ( string, FileRights )

  int addFile(string name) {
    FileRights fr = new FileRights;
    fr.create(name);
    files.add(name,fr);
    0
 }

  File getFile(String name) {
    FileRights fr = files.get(name);
    fr.getFile()
  }

  Auth getAuth(string user, string filename) {
    FileRights fr = files.get(filename);
    fr.getUserAuth(user)
  }

  int addUser_R (string user, string file) {
    FileRights fr = files.get(file);
    Auth readOnly = fr.getR();
    fr.addUserRight(user,readOnly);
    0
  }

  int addUser_W (string user, string file) {
    FileRghts fr = files.get(file);
    Auth writeOnly = fr.getW();
    fr.addUserRight(user,writeOnly);
    0
  }

  int addUser_RW (string user, string file) {
    FileRghts fr = files.get(file);
    Auth readWrite = fr.getRW();
```

```
    fr.addUserRight(user,readWrite);
    0
  }

  int addUser_N (string user, string file) {
    FileRghts fr = files.get(file);
    Auth noRights = fr.getN();
    fr.addUserRight(user,noRights);
    0
  }
}
```

3.6.3 Admin and FileSystem Managers

An Admin object is the owner of a file system. It creates authorizations to access the file system that reflect three file system usage: getting a file, adding a file and associating privileges to users. We could had another layer of protection that would associate these authorizations to different users (with passwords), but the example is sufficient as it is.

```
class Admin : {*} {

  FileSystem fs = new FileSystem;

  Auth gets = authorization( fs, { (getFile + getAuth)* } );
  Auth addFiles = authorization( fs, { addFile* } );
  Auth addRights = authorization( fs,
    { (addUser_R + addUser_W + addUser_RW + addUser_N)* } );

  Auth FSAuth_gets() { gets }
  Auth FSAuth_files() { addFiles }
  Auth FSAuth_rights() { addRights }
}
```

We can now create classes that add files and associate authorizations to users. The following example adds two files (with "from" and "to") to a given file system, using an authorization provided by the given admin. Note that if this admin is not the file system's admin, the authorize method will fail, since the authorization was created to another object. As the desired policy (`{addFile ; addFile }`) is a subset of the authorization policy (`{ addFile*}`), the authorize primitive succeeds and the type system verifies if the code complies with the policy, i.e., there were two files added.

```
class FilesAdder {
```

```
    int AddFiles(Admin admin, FileSystem fs) {

      Auth auth = admin.FSAuth_files();
      authorize fs : auth
      case { addFile ; addFile } : {
        fs.addFile("from");
        fs.addFile("to");
        0
      }
      case error : { 0 }
    }
  }
```

The next example associates the user with name "copyid" to two files named "from" and "to". The method has similar behavior as the previous one.

```
  class RightsAdder {

    int AddRights(Admin admin, FileSystem fs) {

      Auth auth = admin.FSAuth_files();
      authorize fs : auth
      case { (addUser_R + addUser_W + addUser_RW + addUser_N)* } : {
        fs.addUser_R("copyid", "from");
        fs.addUser_W("copyid", "to");
        0
      }
      case error : { 0 }
    }
  }
```

Both this examples can be used to fill the file system with two files ("from" and "to") and create authorizations to that file for the user "copyid".

3.6.4 User Example - CopyFromTo

The goal of the following file system user example is to copy the content of file named "from" to the file named "to". To that end, the user must follow some steps. First, he must ask the file system admin for the authorization to access the file system in order to get files and authorizations. Second, he must apply that authorization, gaining access to the respective file system methods. Third, he must ask the file system for both files and his authorizations over that files. Fourth, he tries to apply both authorizations with the expected policies. If they succeed, he can finally access the file, following the established protocol.

```
class CopyFromTo {

  FileSystem fs;
  Admin admin;

  int init (FileSystem fs, Admin admin) {
    this.fs = fs;
    this.admin = admin;
  }

  int doIt() {
    Auth gets = admin.getFSAuth();

    authorize fs : gets
    case { (getFile+getAuth)* } : {

      File from = fs.getFile( "from" );
      Auth a_from = fs.getAuth( "copyid", "from" );

      File to = fs.getFile( "to" );
      Auth a_to = fs.getAuth( "copyid", "to" );

      authorize from : a_from
      case { open ; read* ; close } : {
        authorize from:a_to
        case { open ; write* ; close } : {
           from.open();
           to.open();
           String s = from.read();
           to.write(s);
           from.close();
           to.close();
           0
        }
        case error : { 0 }
      }
      case error : { 0 }
    }
    case error : { 0 }
  }
}
```

4. The Language and Type System

In this chapter we formalize our language and type system. We begin by presenting the syntax for the language and its operational semantics. Then, we present the type system and prove type safety with a soundness theorem.

4.1 Introduction

The main goal of this dissertation is to develop a type system for access control to objects. Hence, we adapt a simple object-oriented language called ClassicJava [24] to study the concepts introduced in this thesis. However, we simplify even further the language removing inheritance, multiple method parameters and the null value. We then extend the adapted language with the two new primitives to create and apply authorizations as explain in previous chapters.

The result is a simple but sufficient language to demonstrate the use of authorizations and to apply the idealized type system.

4.2 Syntax

We first present the syntax for our language.

Let *Var* be a set ofq denumerable variables denoted by x, y, z and let *Loc* be a set of denumerable memory locations noted by the symbols l, l_0, l_i and $Users$ be a set of denumerable user views denoted by u, u_1, u_i.

Definition 4.2.1. *Terms. Our language is defined by the abstract syntax in figure 4.1.*

A program is a set of class declarations and a main expression as the entry point of the program. A class contains a set of mutable fields with initial values, a set of methods and a maximal class policy declaration. We allow only one method parameter to simplify the reduction and typing rules, though we could add additionally parameters without any consequences to this work's results.

An expression is either a field access, a field update, a method call, a variable binding, a natural number, the reserved word *this* as the self reference, a variable or the new primitives *authorization* and *authorize* to create and apply authorizations, respectively.

Definition 4.2.2. *Values. The set of values Val is defined as follows:*

$$
\begin{array}{rcll}
v & ::= & & (\textit{values}) \\
 & & n & (\textit{natural number}) \\
 & | & u & (\textit{user view}) \\
 & | & \textit{this} & (\textit{self user view}) \\
 & | & a & (\textit{authorization})
\end{array}
$$

P	::=	$defn* \ e$	(*program*)
$defn$	::=	*class cn* : π *body*	(*class decl*)
body	::=	$\{ field* \ method* \}$	(*class body*)
field	::=	$T \ fd = e$	(*field decl*)
method	::=	$T \ m(arg) \ \{ \ e \ \}$	(*method decl*)
arg	::=	$T \ x$	(*variable decl*)
e	::=	*new cn*	(*object creation*)
	|	$e.fd$	(*field access*)
	|	$e.fd = e$	(*field update*)
	|	$e.m(e)$	(*method call*)
	|	*let x = e in e*	(*variable binding*)
	|	n	(*natural number*)
	|	*this*	(*self reference*)
	|	x	(*variable*)
	|	$authorization(e, \pi)$	(*authorization creation*)
	|	*authorize e* : *e*	(*application of authorization*)
		case $\{\pi\}$ *:* $\{e\}$	
		case error: $\{e\}$	
cn	$\in$	*class names*	
fd	$\in$	*field names*	
m	$\in$	*method names*	
x	$\in$	*variable names*	
a	$\in$	*authorization names*	
π	$\in$	*policies*	

Figure 4.1 Abstract syntax for the language.

The values of our language differ from the traditional languages in two aspects. First, we do not use locations as values. Though they are also created, they are only accessible through user views. Second, we add the authorization value for representing authorizations. As such, the values for our language are natural numbers (representing all basic types), user views, the special user view *this* and authorizations. Both user views and authorizations only appear during evaluation time when variables are evaluated.

Definition 4.2.3. *Types. The types used in the syntax of our language are defined as follows:*

T, V	::=	*int*	(*integer type*)
	|	*cn*	(*class type*)
	|	*Auth*	(*authorization type*)

The types used in the syntax are nominal types, meaning that we use classes names to represent types. In addition to class type there are integer type *int* (representing all basic types) and the new authorization type *Auth*.

We now present notations for basic functions used in other definitions. The first is the function $FV(e)$ to determine the free variables of an expression.

Definition 4.2.4. *Free Variables. We define the set of Free Variables of an expression e, $FV(e)$, as follows:*

$$\begin{array}{rcl}
FV(x) & = & \{x\} \\
FV(u) & = & \emptyset \\
FV(a) & = & \emptyset \\
FV(this) & = & \emptyset \\
FV(new\ c) & = & \emptyset \\
FV(e.fd) & = & FV(e) \\
FV(e_1.fd = e_2) & = & FV(e_1) \cup FV(e_2) \\
FV(e_1.m(e_2)) & = & FV(e_1) \cup FV(e_2) \\
FV(let\ x = e_1\ in\ e_2) & = & FV(e_1) \cup (FV(e_2) \setminus \{x\}) \\
FV(authorization(e, \pi)) & = & FV(e) \\
FV\left(\begin{array}{l} authorize\ e_1 : e_2 \\ case\ \pi : \{e_3\} \\ case\ error: \{e_4\} \end{array}\right) & = & FV(e_1) \cup FV(e_2) \cup FV(e_3) \cup FV(e_4)
\end{array}$$

The second is similar, the function $FU(e)$ to determine the free user views of an expression.

Definition 4.2.5. *Free User Views. We define the set of Free User Views of an expression e, $FU(e)$, as follows:*

$$\begin{array}{rcl}
FU(x) & = & \emptyset \\
FU(u) & = & \{u\} \\
FU(a) & = & \emptyset \\
FU(this) & = & \{this\} \\
FU(new\ c) & = & \emptyset \\
FU(e.fd) & = & FU(e) \\
FU(e_1.fd = e_2) & = & FU(e_1) \cup FU(e_2) \\
FU(e_1.m(e_2)) & = & FL(e_1) \cup FL(e_2) \\
FU(let\ x = e_1\ in\ e_2) & = & FU(e_1) \cup FU(e_2) \\
FU(authorization(e, \pi)) & = & FU(e) \\
FU\left(\begin{array}{l} authorize\ e_1 : e_2 \\ case\ \pi : \{e_3\} \\ case\ error: \{e_4\} \end{array}\right) & = & FU(e_1) \cup FU(e_2) \cup FU(e_3) \cup FU(e_4)
\end{array}$$

With the new authorization value, a third function similar to the previous ones is necessary. It is a function to determine the occurrences of authorizations in an expression.

Definition 4.2.6. *Authorization Occurrences. We define the set of authorizations occurrences in an expression e, $FA(e)$, as follows:*

$$
\begin{array}{rcl}
FA(x) & = & \emptyset \\
FA(u) & = & \emptyset \\
FA(a) & = & \{a\} \\
FA(this) & = & \emptyset \\
FA(new\ c) & = & \emptyset \\
FA(e.fd) & = & FA(e) \\
FA(e_1.fd = e_2) & = & FA(e_1) \cup FA(e_2) \\
FA(e_1.m(e_2)) & = & FA(e_1) \cup FA(e_2) \\
FA(let\ x = e_1\ in\ e_2) & = & FA(e_1) \cup FA(e_2) \\
FA(authorization(e, \pi)) & = & FA(e) \\
FA\left(\begin{array}{l} authorize\ e_1 : e_2 \\ case\ \pi : \{e_3\} \\ case\ error: \{e_4\} \end{array}\right) & = & FA(e_1) \cup FA(e_2) \cup FA(e_3) \cup FA(e_4)
\end{array}
$$

Finally, we define capture-avoiding substitution on expressions in the expected way.

Definition 4.2.7. *Substitution. We define capture-avoiding substitution of a user view u for free occurrences a variable x (or another user view u_1) in an expression e, written $e\{^u/_x\}$ (respectively, $e\{^u/_{u_1}\}$), as follows:*

$$
\begin{array}{rcl}
new\ c\{^u/_x\} & = & new\ c \\
e.fd\{^u/_x\} & = & e\{^u/_x\}.fd \\
(e_1.fd = e_1)\{^u/_x\} & = & e_1\{^u/_x\}.fd = e_2\{^u/_x\} \\
e_1.m(e_1)\{^u/_x\} & = & e_1\{^u/_x\}.m(e_2\{^u/_x\})
\end{array}
$$

$$
(let\ y = e_1\ in\ e_2)\{^u/_x\} = \begin{cases} let\ y = e_1\{^u/_x\}\ in\ e_2\{^u/_x\} & \text{, if } x \neq y \text{ and } y \neq u \\ let\ y = e_1\{^u/_x\}\ in\ e_2 & \text{, if } x = y \end{cases}
$$

$$
y\{^u/_x\} = \begin{cases} u & \text{, if } x = y \\ y & \text{, if } x \neq y \end{cases}
$$

$$
this\{^u/_x\} = \begin{cases} u & \text{, if } x = this \\ this & \text{, if } x \neq this \end{cases}
$$

$$
\left(\begin{array}{l} authorize\ e_1 : e_2 \\ case\ \pi : \{e_3\} \\ case\ error: \{e_4\} \end{array}\right)\{^u/_x\} = \left(\begin{array}{l} authorize\ e_1\{^u/_x\} : e_2\{^u/_x\} \\ case\ \pi : \{e_3\{^u/_x\}\} \\ case\ error: \{e_4\{^u/_x\}\} \end{array}\right)
$$

$$
authorization(e, \pi)\{^u/_x\} = authorization(e\{^u/_x\}, \pi)
$$

Capture-avoiding substitution is undefined if the conditions are not met and can be avoided by renaming of bound variables.

4.3 Operational Semantics

We now define the operation semantics for our language. To this end, we must define some concepts needed by the reduction rules.

First, we need to define what locations contain. Since only object's fields are mutable, locations will only hold objects. We will simplify the rules by omitting the integer value and only considering authorizations when they are expected. With this restrictions, variables and fields will only be associated with user views, allowing a focus on their policies.

Definition 4.3.1. *Object Location. An object location contains two components: a set F to associate field names with their values and a set M store the object methods information.*

$$\begin{array}{lcll} F & ::= & \emptyset & \\ & | & F, fd \rightarrow v & (\textit{field}) \\ \\ M & ::= & \emptyset & \\ & | & M, m(x) = e & (\textit{method info}) \end{array}$$

We can now define the store used on the reduction rules. The store contains the object locations in run-time and the program classes information. However, the access to those locations are through user views, protecting them with a policy. Additionally, we have authorizations to manage the policies.

Definition 4.3.2. *Store. A store S is an association between run-time elements and their values. Its domain is program classes information, locations, user views and authorizations.*

$$\begin{array}{lcll} S & ::= & \emptyset & \\ & | & S, c \rightarrow \{F \;\; M \;\; \pi\} & (\textit{class information}) \\ & | & S, u \rightarrow (l, \; \pi) & (\textit{user view}) \\ & | & S, l \rightarrow \{F \;\; M\} & (\textit{location}) \\ & | & S, a \rightarrow [l, \; \pi] & (\textit{authorization}) \end{array}$$

We now define the operations performed with stores.

Definition 4.3.3. *Store Operations. We define the following operations on stores:*

(*Domain*). We say $Dom(S)$ to denote the domain of S.

(*Information of a Class*). We say $S(c)$ to denote the information associated with class c in store S.

if $S = \{..., \; c \rightarrow \{F \;\; M \;\; \pi\}, \; ...\}$
then $S(c) = \{F \;\; M \;\; \pi\}$,
otherwise $S(c)$ *is undefined.*

(*Value of an Element*). We say $S(x)$ to denote the value associated with element x in store S.

if $S = \{..., x \to v, ...\}$

then $S(x) = v$,

otherwise $S(x)$ *is undefined.*

(*Update of a User View Policy*). We say $S[u \leftarrow (l, \pi)]$ to denote the policy modification of user view u to the policy π.

if $S = \{..., u \to (l, \pi'), ...\}$

then $S[u \leftarrow (l, \pi)] = \{..., u \to (l, \pi), ...\}$,

otherwise $S[u \leftarrow (l, \pi)]$ *is undefined.*

(*Update of a Field*). We say $S[u.fd \leftarrow v]$ to denote the value modification of field fd of the location associated by user view u.

if $S = \{..., u \to (l, \pi), ...\}$ *and* $S(l) = \{F \ M\}$ *and* $F = \{..., fd \to v', ...\}$

then $S[u.fd \leftarrow v] = \{..., u \to (l, \pi), ...\}$ *where* $S(l) = \{F \ M\}$ *and* $F = \{..., fd \to v, ...\}$

otherwise $S[u.fd \leftarrow v]$ *is undefined.*

We now define simple relations between store and expressions: closed expressions and valid configurations.

Definition 4.3.4. *Store and Expressions relations. We define the following relations between a store and expressions:*

(*Closed Expression*). An expression e is closed in a store S if all user views and authorizations occurring in e are elements of $Dom(S)$ and e does not have free variables.

e *is closed in* $S \triangleq FU(e) \cup FA(e) \in Dom(S) \wedge FV(e) = \emptyset$.

(*Valid Configuration*). We say that the pair $< S; e >$ is a valid configuration if expression e is closed in store S.

$< S; e >$ *is a valid configuration* $\triangleq e$ *is closed in* S.

Finally, to apply authorizations we need to define an operation that verifies if a pair (location, policy) is valid with the authorization, meaning that the authorization is applicable to the desired pair.

Definition 4.3.5. *Authorization Applicability.We define authorization applicability, which validates if an authorization* $[y, \pi']$ *can be applied to a location* l *and policy* π*, written* $[y, \pi'] \vdash [l, \pi]$*, as following:*

$$[y, \pi'] \vdash [l, \pi] \triangleq l = y \wedge \pi \subseteq \pi'$$
$$[y, \pi'] \nvdash [l, \pi] \triangleq l \neq y \vee \pi \not\subseteq \pi'$$

4.3.1 Reduction Rules

We are now ready to define the semantics of our language. To do so we use a small step operational semantics, following a call-by-value evaluation strategy.

Let S be a store and e an expression such that $< S; e >$ is a valid configuration, a reduction rule is a single step evaluation of e to an expression e', where the resulting store is S'. The judgement has the following form:

$$< S \ ; \ e > \longrightarrow < S' \ ; \ e' >$$

The operational semantics of the language is defined inductively by the following rules:

Object Creation. By creating a new object, a new location and a new user view with the maximal class privileges to that location are generated. Thus, it produces no effects in existing user views policies.

[R - Object Creation]

$$\frac{l, u \notin Dom(S) \qquad c \rightarrow \{F \ \ M \ \ \pi\} \in S}{< S \ ; \ new \ c \ > \longrightarrow < S' \ ; \ u >}$$

$$S' = S \uplus \{l \rightarrow \{F \ \ M\}, u \rightarrow (l, \ \pi)\}$$

Field Access. As we have seen, a field access generates a new user view to the same location and with the same policy. As a consequence, the field loses its privileges to that location.

[R - Field Access]

$$\frac{\begin{array}{ll} u \rightarrow (l, \ \pi_l) \in S & l \rightarrow \{F \ \ M\} \in S \\ u' \notin Dom(S) & fd \rightarrow (l', \ \pi) \in F \end{array}}{< S \ ; \ u.fd > \longrightarrow < S' \ ; \ u' >}$$

$$S' = S[u.fd \leftarrow (l', \ \emptyset)] \uplus \{u' \rightarrow (l', \ \pi)\}$$

Field Update. While the right side expression is not a value, a field update will not produce any extra effect besides the ones of the reduction of that expression.

[R - Field Update]

$$\frac{< S \ ; \ e > \longrightarrow < S' \ ; \ e' >}{< S \ ; \ u.fd = e > \longrightarrow < S' \ ; \ u.fd = e' >}$$

Field Update - Base. Updating a field with another user view changes both location and policy of such field with the same elements of the user view (location and policy). As a consequence, that user view loses its privileges to that location.

[R - Field Update - Base]

$$\frac{\begin{array}{ll} u \to (l,\ \pi) \in S & l \to \{F\ \ M\} \in S \\ u' \to (l',\ \pi') \in S & fd \in Dom(F) \end{array}}{< S\ ;\ u.fd = u' > \longrightarrow < S'\ ;\ u' >}$$

$$S' = S[u.fd \leftarrow (l',\ \pi'), u' \leftarrow (l',\ \emptyset)]$$

Method Call. While the argument expression is not a value, a method call will not produce any extra effect besides the ones of the reduction of argument's expression.

[R - Method Call]

$$\frac{< S\ ;\ e > \longrightarrow < S'\ ;\ e' >}{< S\ ;\ u.m(e) > \longrightarrow < S'\ ;\ u.m(e') >}$$

Method Call - Base. Calling a method is like entering an administrator user space: we have unrestricted access to that object. As such, a new view is generated with an unrestricted policy over the location and will substitute the *this* keyword within the method body. Another user view with an empty policy is also generated for the location of the argument and will substitute the parameter of that method. Note that the argument will not lose its policy. The effect of calling a method of an object is to take a step in the policy according to the method name (third property of a policy language).

[R - Method Call - Base]

$$\frac{\begin{array}{lll} u_1 \to (l,\ \pi) \in S & l \to \{F\ \ M\} \in S & u_3, u_t \notin Dom(S) \\ u_2 \to (l_u,\ \pi_u) \in S & m(x) = e \in M & \pi \xrightarrow{m} \pi' \end{array}}{< S\ ;\ u_1.m(u_2) > \longrightarrow < S'\ ;\ e\{^{u_3}/_{x}\}\{^{u_t}/_{this}\} >}$$

$$S' = S[u_1 \leftarrow (l,\ \pi')] \uplus \{u_3 \to (l_u,\ \emptyset), u_t \to (l,\ *)\}$$

Variable Binding. While the left side expression is not a value, a variable binding will not produce any extra effect besides the ones of the reduction of that expression.

[R - Variable Binding]

$$\frac{< S\ ;\ e_1 > \longrightarrow < S'\ ;\ e_1' >}{< S\ ;\ let\ x = e_1\ in\ e_2 > \longrightarrow < S'\ ;\ let\ x = e_1'\ in\ e_2 >}$$

Variable Binding - Base. When the left side expression is a user view, the variable binding reduces to the right side expression where the variable is substituted with a new user view with the same privileges of the first user view, which loses its privileges.

[R - Variable Binding - Base]

$$\frac{u \rightarrow (l,\ \pi) \in S \qquad u_2 \notin Dom(S)}{< S\ ;\ let\ x = u\ in\ e_2 > \longrightarrow < S'\ ;\ e_2\{^{u_2}/_x\} >}$$

$$S' = S[u \leftarrow (l,\ \emptyset)] \uplus \{u_2 \rightarrow (l,\ \pi)\}$$

Authorization. Creating an authorization has no effect in any policy. The only restriction is that the new policy must be a subset of the current policy. One can only create authorizations with equal or less privileges than his own.

[R - Authorization]

$$\frac{u \rightarrow (l,\ \pi_u) \in S \qquad \pi <: \pi_u \qquad a \notin Dom(S)}{< S\ ;\ authorization(u, \pi) > \longrightarrow < S'\ ;\ a >}$$

$$S' = S \uplus \{a \rightarrow [l,\ \pi]\}$$

Authorize OK. An application of an authorization succeeds if its information validates the pair (location, policy), meaning the location is the same and the expected policy is a subset of the policy stored in that authorization. So, the user view's policy is changed to the expected one and the computation continues with the first case expression.

[R - Authorize OK]

$$\frac{u \rightarrow (l,\ \pi_u) \in S \qquad S(a) \vdash [l,\ \pi]}{< S\ ;\ \begin{array}{l} authorize\ u : a \\ case\ \pi :\ \{e_1\} \\ case\ error:\ \{e_2\} \end{array} > \longrightarrow < S'\ ;\ e_1 >}$$

$$S' = S[u \leftarrow (l,\ \pi)]$$

Authorize KO. An application of an authorization fails if its information does not validate the pair (location, policy), meaning either the location is different or the given policy and the one stored in that authorization are inconsistent. When this happens, the *authorize* reduces to the case error expression without any changes.

[R - Authorize KO]

$$\frac{u \to (l,\, \pi_u) \in S \qquad S(a) \nvdash [l,\, \pi]}{\begin{array}{l} \textit{authorize } u : a \\ < S\,; \quad \textit{case } \pi : \{e_1\} \qquad > \longrightarrow < S\,;\, e_2 > \\ \textit{case error: } \{e_2\} \end{array}}$$

4.4 Type System

We now define the type system. We start by presenting the types for our language.

Definition 4.4.1. *Types. The types for our language are defined as follows:*

$$\begin{array}{rcll} T,V & ::= & O\{\pi\} & (\textit{user type}) \\ & | & \textit{Auth} & (\textit{authorization type}) \\ & | & \textit{int} & (\textit{Integer Type}) \end{array}$$

A user type $O\{\pi\}$ is composed by an object type O defined below and the current policy π for that object. The types *Auth* and *int* are to be associated with authorizations and integer values, respectively.

Definition 4.4.2. *Object Type. An object type is defined as follows:*

$$\begin{array}{rcll} F & ::= & \emptyset & \\ & | & F, fd : T & (\textit{field type}) \\ \\ M,N & ::= & \emptyset & \\ & | & M, m : T(V) & (\textit{method type}) \\ \\ O & ::= & & (\textit{Object Type}) \\ & & M & (\textit{External Object Type}) \\ & | & [F\ \ M] & (\textit{Internal Object Type}) \end{array}$$

An object type can be either external, to type normal variables and user views, where field types are not visible, or internal, to type variables and user views alias of the self reference *this*, where fields are visible. Thus, an external object type is composed by a set of method names

associated with their method type $T(V)$, where T is the return type and V is the parameter type. An internal object type also contains this set in addition to a set of field names associated with their types.

Types are associated to variables, user views and authorizations on a typing environment, which we now define.

Definition 4.4.3. *Typing Environment. The set off all possible typing environments is defined as follows:*

$$
\begin{array}{lcll}
\Delta & ::= & \emptyset & \\
 & | & \Delta, c : [F \;\; M \;\; \pi] & (\textit{class information}) \\
 & | & \Delta, x : T & (\textit{variable}) \\
 & | & \Delta, \textit{this} : [F \;\; M]\{*\} & (\textit{self reference}) \\
 & | & \Delta, a : \textit{Auth} & (\textit{authorization}) \\
 & | & \Delta, u : O\{\pi\} & (\textit{user view})
\end{array}
$$

A typing environment can contain: the program classes type information associated with their names; variables associated with their respective type; the reserved keyword *this* associated with an internal object type and unrestricted policy $*$; authorizations associated with the reserved type *Auth*; and user views associated with their object type and their current policy π.

We can now define some operations with type environments.

Definition 4.4.4. *Typing Environment Operations. We use the standard notations to define the domain of a typing environment ($Dom(\Delta)$) and the function to return the type associated with an element ($\Delta(x)$). To compare two typing environments, we also define the standard subset relation ($\subset$) and a special subset relation ($\subset^{\pi}$) that ignores user view policies, since we will need to compare environments from different points of the program and the policies of common user views may vary. We call the second relation π-subset.*

(*Domain*). We say $Dom(\Delta)$ to denote the domain of Δ.

(*Type of an element*). We say $\Delta(x)$ to denote the type associated with an element x in typing environment Δ.

if $\Delta = \{..., x : T, ...\}$ *then* $\Delta(x) = T$, *otherwise* $\Delta(x)$ *is undefined.*

(*Subset*). Let Δ_1 and Δ_2 be typing environments, Δ_1 is a subset of Δ_2 if all elements of Δ_1 are elements of Δ_2.

$$\Delta_1 \subseteq \Delta_2 \triangleq \forall\, i \in \Delta_1 \Rightarrow i \in \Delta_2$$

(π-*subset*). Let Δ_1 and Δ_2 be typing environments, Δ_1 is a π-subset of Δ_2 if all elements of Δ_1 except user views are elements of Δ_2. For each user views of Δ_1 there must be a user view in Δ_2 with the same name and object type, but the policy may be different.

$$\Delta_1 \subseteq^{\pi} \Delta_2 \triangleq \forall\, u \rightarrow O\{\pi_1\} \in \Delta_1 \Rightarrow u \rightarrow O\{\pi_2\} \in \Delta_2$$

(*Policy Update*). We say $\Delta[id \leftarrow \pi]$ to denote the policy modification of user view or variable id to the policy π.

$$\begin{aligned}
&\textit{if } \Delta = \{..., id \rightarrow O\{\pi'\}, ...\} \\
&\textit{then } \Delta[id \leftarrow \pi] = \{..., id \rightarrow O\{\pi\}, ...\}, \\
&\textit{otherwise } \Delta[id \leftarrow \pi] \textit{ is undefined.}
\end{aligned}$$

We now define the necessary relations between typing environment and expressions.

Definition 4.4.5. *Typing Environment and Expressions relations. We define closed expression and typing judgement as follows:*

(*Closed Expression*). An expression e is closed in a typing environment Δ if all free elements occurring in e are elements of $Dom(\Delta)$.

$$e \textit{ is closed in } \Delta \triangleq FV(e) \cup FU(e) \cup FA(e) \in Dom(\Delta).$$

(*Typing Judgement*). Let Δ be a typing environment, e an expression, such that e is closed in Δ, and T a type, a typing judgement takes the form $\Delta \vdash e : T \mapsto \Delta'$ and asserts that expression e has type T with relation to the typing environment Δ, producing Δ' as result of the effects. As such, we can say that Δ is a π-subset of Δ', $\Delta \subseteq^{\pi} \Delta'$, since effects only target policies.

Finally, we define the relation between a store and a typing environment.

Definition 4.4.6. *Well-typed store. We say that a store S is well-typed with respect to a typing environment Δ, written $\Delta \vdash S$, if $Dom(S) = Dom(\Delta)$ and $\forall x \in Dom(S) \Rightarrow \Delta \vdash S(x) : \Delta(x)$.*

4.4.1 Typing Rules

As before, we will simplify the rules by considering that expressions and variables are only typed by user types, allowing a focus on policies evolution. Authorization types will also be considered when expected.

Our language typing rules are defined below with the following form:

[T - Rule Name]

$$\frac{\textit{premises}}{\Delta \vdash e : T \mapsto \Delta'}$$

Each rule has some judgements as premises and a judgement as conclusion. When the number of premises is zero, the rule is called an *axiom*. A typing rule states that when all premises are valid, so is the conclusion.

Object Creation. An object creation is typed with a user type, where the method type and initial policy are obtained by the class information. Since the object is new, the user view has the maximal class policy defined in the class declaration. Typing an object creation has no effect on existing policies.

[T - Object Creation]

$$\frac{c : [F \;\; M \;\; \pi] \in \Delta}{\Delta \vdash new\ c : M\{\pi\} \mapsto \Delta}$$

Authorization Value. A authorization is typed with the *Auth* type.

[T - Authorization Value]

$$\frac{a : Auth \in \Delta}{\Delta \vdash a : Auth \mapsto \Delta}$$

Variable and User View Access. Accessing a variable or user view (id, for short) will have one of two consequences, depending on where it is being accessed. It can either change the policy of the id to an empty one and the expression is typed with the current policy, or it can have no effect on the id's policy and the expression is typed with an empty policy.

[T - id - 1] [T - id - 2]

$$\frac{id : M\{\pi\} \in \Delta}{\Delta \vdash id : M\{\emptyset\} \mapsto \Delta} \qquad \frac{id : M\{\pi\} \in \Delta}{\Delta \vdash id : M\{\pi\} \mapsto \Delta[id \leftarrow \emptyset]}$$

Note that these rules could be joint together to create a more generic rule, where we allow a *decomposition* of a policy into two sub-policies, written $\pi >> \pi_1, \pi_2$:

[T - id]

$$\frac{\pi >> \pi_1, \pi_2}{\Delta, id : M\{\pi\} \vdash id : M\{\pi_1\} \mapsto \Delta, id : M\{\pi_2\}}$$

We will not use this final rule since we consider it to be irrelevant for our goals.

This. The self reference user view is also typed with user types, but it has no effect on the environment, i.e., the self reference type remains the same (unrestricted privileges). This reflects the analogy with administrator space, where any operation can be performed. However, we still need two rules as in the previous case.

[T - This - 1] [T - This - 2]

$$\frac{this : [F \;\; M]\{*\} \in \Delta}{\Delta \vdash this : [F \;\; M]\{\emptyset\} \mapsto \Delta} \qquad \frac{this : [F \;\; M]\{*\} \in \Delta}{\Delta \vdash this : [F \;\; M]\{*\} \mapsto \Delta}$$

Field Access. As in the previous cases, variable or user views' (id) field accesses are either typed with the same user type as the field, who will lose its policy, or with a user type with an empty policy, with no changes to the field's type.

[T - Field Access - 1] [T - Field Access - 2]

$$\frac{\begin{array}{c} id : [F \;\; M]\{\pi\} \in \Delta \\ fd : M'\{\pi'\} \in F \end{array}}{\Delta \vdash id.fd : M'\{\emptyset\} \mapsto \Delta} \qquad \frac{\begin{array}{c} id : [F \;\; M]\{\pi\} \in \Delta \\ fd : M'\{\pi'\} \in F \end{array}}{\Delta \vdash id.fd : M'\{\pi'\} \mapsto \Delta[id.fd \leftarrow \emptyset]}$$

Field Update. Variable or user views' (id) field update are typed with a user view with the same object type as the expression, which is the same as the field, but with an empty policy. The field's policy is changed to the from the expression's user type.

[T - Field Update]

$$\frac{id : [F \;\; M]\{\pi\} \in \Delta \qquad fd : M'\{\pi'\} \in F \qquad \Delta \vdash e : M'\{\pi_e\} \mapsto \Delta'}{\Delta \vdash id.fd = e : M'\{\emptyset\} \mapsto \Delta'[id.fd \leftarrow \pi_e]}$$

Variable Binding. The type of a variable binding is obtained as usual. The left side expression is typed with user type in the initial environment, producing an effect over it and creating a second environment. The right side expression is typed with another user type in the new environment plus the parameter typed with the first user type, producing a third environment, result of applying the effects over the second environment. Thence, after typing a variable binding, the third environment becomes available as the effect of the expression.

[T - Variable Binding]

$$\frac{\Delta_1 \vdash e_1 : M_1\{\pi_1\} \mapsto \Delta_2 \quad \Delta_2, x : M_1\{\pi_1\} \vdash e_2 : M_2\{\pi_2\} \mapsto \Delta_3, x : M_1\{\pi_3\}}{\Delta_1 \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : M_2\{\pi_2\} \mapsto \Delta_3}$$

Method Call. The typing of an variable or user view's (id) method call is one of the most important rules of this type system. First, a method call is typed with a user type with the same object type as the method's result and with an empty policy. Second, the argument expression is typed with an user view with the same object type as the method's parameter plus an empty policy. As such, when calling a method with an object as an argument, one does not lose the privileges associated with that object. Third, the id must exist in both environments (before and after argument expression typing), but it can have different policies (before-policy and after-policy). The before-policy is ignored, since it was already considered by the argument's expression typing. To the method call be well-typed, the after-policy must be able to take a step to another policy using the method name (third policy language property). If it does, the typing will succeed, changing the object's current policy to the after-policy.

[T - Method Call]

$$\frac{\begin{array}{ll}\Delta \vdash e : M_2\{\emptyset\} \mapsto \Delta' & m : M_1(M_2) \in M \\ id : M\{\pi_a\} \in \Delta & \pi_b \xrightarrow{m} \pi \\ id : M\{\pi_b\} \in \Delta' & \end{array}}{\Delta \vdash id.m(e) : M_1\{\emptyset\} \mapsto \Delta'[x \leftarrow \pi]}$$

Authorization. The creation of an authorization is typed with the *Auth* type if the desired policy is a subtype of the indicated variable or user view's policy. As such, authorizations can only be created with less or equal privileges as the current owned ones.

[T - Authorization]

$$\frac{id : M\{\pi_{id}\} \in \Delta \qquad \pi <: \pi_{id} \qquad M \vdash \pi}{\Delta \vdash \mathit{authorization}(id, \pi) : \mathit{Auth} \mapsto \Delta}$$

Authorize. The typing of application of authorizations always succeeds as long as we are applying an existing authorization to an existing variable or user view (id) and the expected policy is valid with the id's object type. It will be typed with the same type and producing the same environment as both case expressions. As such, whether the application was successful or not, both typing of case expressions must produce the same environment, even beginning with different policies for the id.

[T - Authorize]

$$\frac{\begin{array}{ll} id : M\{\pi_{id}\} \in \Delta & \Delta[id \leftarrow \pi] \vdash e_1 : T \mapsto \Delta' \\ a : Auth \in \Delta & \Delta \vdash e_2 : T \mapsto \Delta' \\ M \vdash \pi & \end{array}}{\Delta \vdash \begin{array}{l} authorize\ id : a \\ case\ \pi : \{e_1\} \\ case\ error: \{e_2\} \end{array} : T \mapsto \Delta'}$$

Class. As usual, a class is well-typed if all fields' initial expression have the expected type and each method's body is typed with the return type declared for that method. However, due to the use of policies, we need to perform more checks. The initial policy for each field is the respective expression's resulting policy. With that information, plus the declared method types, we can create an environment with the self reference *this* on which methods' bodies can be typechecked. As we have seen, field's policies are invariant. So, after typing a method's body, each field's policy must be the same as their initial policy, i.e., the environment remains the same before and after the typing of a method.

[T - Class]

$$\frac{\begin{array}{c} \forall i \in \{1..n\} \Rightarrow \Delta \vdash e_i : T_i\{\pi_i\} \mapsto \Delta \\ F_t = \{fd_i : T_i\{\pi_i\} \mid i \in \{1..n\}\} \\ M_t = \{m_i : M_i(V_i) \mid i \in \{1..k\}\} \\ \forall j \in \{1..k\} \Rightarrow \ let\ \Delta_j = \Delta, this : [F_t\ \ M_t]\{*\}, x_j : N_j\{\emptyset\} \\ \Delta_j \vdash e'_j : M_1\{\emptyset\} \vdash \Delta_j \end{array}}{\begin{array}{l} \Delta \vdash_{\mathrm{c}} \ class\ cn : \pi\{ \\ \qquad\qquad T_1\ fd_1 = e_1\ ...\ T_n\ fd_n = e_n \\ \qquad\qquad M_1\ m_1(N_1\ x_1)\{e'_1\}\ ...\ M_k\ m_k(N_k\ x_k)\{e'_k\} \\ \quad \} \end{array}}$$

4.4.2 Type Safety

We now present the main technical result of this thesis: the type soundness theorem.

Theorem 4.4.1. ***Type Soundness***. *If a program is well-typed, then its evaluation will not get stuck, that is, it will not reach faulty states, particularly unauthorized method calls.*

Proof. Applying both progress (4.4.3) and subject reduction (4.4.2) theorems we get the desired result for this theorem. As such, the combine proofs of both theorem compose the proof for this theorem. □

We need to define the set of forbidden errors (or faulty configuration) - the errors that the type system is able to capture.

Definition 4.4.7. *Faulty Configuration. A faulty configuration is an undefined operation, that is, all cases where the operational semantics gets stuck. The respective configurations are the following:*

Object Creation. An object creation is faulty if the class does not exists.

[F - Object Creation]

$$\frac{c \notin Dom(S)}{< S \; ; \; new \; c \; >}$$

Field Access. A field access is faulty if the value to which it is applied is not a user view or if the field does not exists.

[F - Field Access - not a user view]

$$\frac{v \rightarrow (l, \; \pi) \notin S}{< S \; ; \; v.fd >}$$

[F - Field Access - no field]

$$\frac{u \rightarrow (l, \; \pi_t) \in S \qquad l \rightarrow \{F \;\; M\} \in S \qquad fd \notin Dom(F)}{< S \; ; \; u.fd >}$$

Field Update. A field update is faulty if in the same conditions as the previous case.

[F - Field Update - not a user view]

$$\frac{v \rightarrow (l, \; \pi_t) \notin S}{< S \; ; \; v.fd = e >}$$

[F - Field Update - no field]

$$\frac{u \rightarrow (l, \; \pi_t) \in S \qquad l \rightarrow \{F \;\; M\} \in S \qquad fd \notin Dom(F)}{< S \; ; \; u.fd = e >}$$

Method Call. A method call is faulty if the value to which it is applied is not a user view, the method does not exist or the user view's policy does not allow the call (the method is not visible).

[F - Method Call - not a user view]

$$\frac{v \rightarrow (l,\ \pi_t) \notin S}{< S\ ;\ v.m(e) >}$$

[F - Method Call - no method]

$$\frac{u \rightarrow (l,\ \pi_t) \in S \qquad l \rightarrow \{F\ \ M\} \in S \qquad m \notin Dom(M)}{< S\ ;\ u.m(e) >}$$

[F - Method Call - security policy violation: unauthorized access]

$$\frac{\begin{array}{ll} u \rightarrow (l,\ \pi) \in S & m(x) = e \in M \\ l \rightarrow \{F\ \ M\} \in S & \pi \overset{m}{\nrightarrow} \end{array}}{< S\ ;\ u.m(e) >}$$

Authorization. An authorization creation is faulty if the value to which the authorization is being created is not a user view or the desired policy is not a subset of the current user view's policy.

[F - Authorization - not a user view]

$$\frac{v \rightarrow (l,\ \pi_v) \notin S}{< S\ ;\ authorization(v, \pi) >}$$

[F - Authorization - invalid policy]

$$\frac{u \rightarrow (l,\ \pi_u) \in S \qquad \pi \not<: \pi_u}{< S\ ;\ authorization(u, \pi) >}$$

Authorize. An authorization application is faulty if the value to be applied is not an authorization or the value with which the authorization is being applied is not a user view.

[F - Authorize - not a user view]

$$\frac{v \rightarrow (l,\ \pi_v) \notin S}{\begin{array}{c} authorize\ v : a \\ < S\ ; \quad case\ \pi : \{e_1\} \qquad > \\ \qquad case\ error: \{e_2\} \end{array}}$$

[F - Authorize - not an authorization]

$$\frac{v \rightarrow [l,\, \pi] \notin S}{\begin{array}{l} authorize\ u : v \\ < S\ ;\quad case\ \pi : \{e_1\} \qquad > \\ \qquad\quad case\ error:\ \{e_2\} \end{array}}$$

Configurations that result from faulty configurations are, naturally, also faulty:

Field Update. A field update is faulty also when the right side expression is faulty.

[F2 - Field Update - faulty sub-expression]

$$\frac{< S\ ;\ e >\ is\ faulty}{< S\ ;\ u.fd = e >}$$

Variable Binding. A variable is faulty when either the left-side expression is faulty or the right-side expression is faulty.

[F2 - Variable Binding - faulty left side expression]

$$\frac{< S\ ;\ e_1 >\ is\ faulty}{< S\ ;\ let\ x = e_1\ in\ e_2 >}$$

[F2 - Variable Binding - faulty right side expression]

$$\frac{< S\ ;\ e_1 >\ is\ faulty}{< S\ ;\ let\ x = e_1\ in\ e_2 >}$$

Method Call. A method call is faulty also when the argument expression is faulty.

[F2 - Method Call - faulty argument expression]

$$\frac{< S\ ;\ e >\ is\ faulty}{< S\ ;\ u.m(e) >}$$

Now we are almost ready to present both theorems. To that end, we define two simple lemmas: substitution and weakening. The former states that substitution does not affect typing and the latter states that typing is not affected by trash on the typing environment.

The traditional substitution lemma states that if an expressions is well-typed in an environment containing a variable, it continues to be well typed when the variable is substituted by an expression of the same type. In our case, we have to take into account the user views. As such, variables can only be substituted by user views with the same object type and policy. Furthermore, the user view cannot already exist in the environment or its policy could be violated. The lemma should also be applicable when substituting user views.

Lemma 4.4.1. *Substitution Lemma. Let Δ be a typing environment, x a variable typed with the user type $M\{\pi\}$, and e be typed with T in Δ with x, if there is a user view also typed with the user type $M\{\pi\}$ not belonging to Δ, then variable x can be substituted in e by the user view u without changing any type and effect of e.*

$$\begin{array}{ll} \textit{if} & \Delta, x : M\{\pi\} \vdash e : T \mapsto \Delta', x : M\{\pi'\} \\ \textit{and} & u : M\{\pi\} \notin \Delta \\ \\ \textit{then} & \Delta, u : M\{\pi\} \vdash e\{^{u}/_{x}\} : T \mapsto \Delta', u : M\{\pi'\}. \end{array}$$

Proof. The lemma follows directly by induction on the derivation of $\Delta, x : M\{\pi\} \vdash e : T \mapsto \Delta', x : M\{\pi'\}$. □

The weakening lemma is the same as usual. The additional variables and user views will not be used in the expression and so their policies will not change.

Lemma 4.4.2. *Weakening Lemma. Let Δ be a typing environment and e be typed with T in Δ, if there is a new element in the environment, the judgement remains valid and the expression is typed with the same type. Also, the new element's policy will not suffer any effect.*

$$\begin{array}{ll} \textit{if} & \Delta \vdash e : T \mapsto \Delta' \\ \textit{and} & x \notin \mathit{Dom}(\Delta) \\ \\ \textit{then} & \Delta, x : T' \vdash e : T \mapsto \Delta', x : T'. \end{array}$$

Proof. The lemma also follows directly by induction on the given derivation. We can insert the additional hypothesis in every hypothetical judgment occurring in the derivation without invalidating any rule applications. □

We are now able to define and prove the subject reduction and progress theorems. The former states that if an expression is well-typed and it reduces to another expression, then the second expression is also well-typed with the same type as the first expression, i.e., the type is preserved by reductions. The latter that if an expression is well-typed, then either it is a value or it reduces to another expression.

Theorem 4.4.2. *Subject Reduction. Let Δ and Δ'' be a typing environments, e an expression such that e is closed in Δ, and T a type.*

$$\begin{array}{ll} \textit{if} & \Delta \vdash e : T \mapsto \Delta''(e \textit{ is closed in } \Delta) \\ \textit{and} & \exists\, S \,:\, \Delta \vdash S \\ \textit{and} & < S \,;\, e > \longrightarrow < S' \,;\, e' > \\ \\ \textit{then} & \exists\, \Delta', \Delta'' \,:\, \Delta' \vdash S',\ \Delta \subseteq^{\pi} \Delta',\ \Delta'' \subseteq \Delta''' \\ \textit{and} & \Delta' \vdash e' : T \mapsto \Delta'''. \end{array}$$

Proof. By induction on the derivation of $< S\ ;\ e > \longrightarrow < S'\ ;\ e' >$. The complete proof is presented in appendix A, page 73. □

Theorem 4.4.3. *Progress. Let Δ and Δ' be typing environments, e an expression such that e is closed in Δ, and T a type.*

if $\quad \Delta \vdash e : T \mapsto \Delta'(e \text{ is closed in } \Delta)$

then $\quad \exists\, S\ :\ \Delta \vdash S \text{ and } < S\ ;\ e > \longrightarrow < S'\ ;\ e' >$
or $\quad e$ *is a value.*

Proof. By induction on the typing derivation. The complete proof is presented in appendix A, page 80. □

4.5 Remarks

In this chapter we have formalized our language and type system in accordance with their informal description presented in chapter 3.

We have seen how both the operational semantics and the type system keep the state of user view's policies. The former needs them to be able to verify if an authorization can be applied (rules [R - Authorize OK] and [R - Authorize KO]) and the latter uses them to statically verify if an authorization can be created and a method call is authorized (rules [T - Authorization] and [T - Method Call]).

We have also seen that new user views are generated when a variable is created and when methods' parameters are instantiated (rules [R - Variable Binding] and [R - Method Call]). Additionally, non-empty policies (i.e., authorized policies) are never shared by two user views (rules [T - id - 1 & 2], [T - Field Access - 1 & 2] and [T - Field Update]). So, each variable will be associated with a unique user view, allowing aliasing without consequences to their policies. Aliasing is also possible since fields' policies are invariant. At the beginning and at the end of each method, the policies of each class' fields must be the same ([T - Class]).

Creating and applying authorizations allow a dynamic control over the objects. When using the authorize primitive, one has to describe the expected policy that the authorization holds. Since that information is only available at run-time, the type system verifies if the expressions for both cases are performed according to their respective policies for the selected user view. The verification to confirm if the expected policy is valid for that user view is performed at run-time.

Taking all this into consideration, we were able to prove the main goal of our work: a type safety theorem stating that a well-typed program does not have unauthorized method calls and so it can be executed without breaking any access control policy to objects.

As an interesting property, our type system treats policies abstractly, using properties of a generic policy language when reasoning about them (rules [T - Method Call], [T - Authorization] and [T - Authorize]). So, we can create different policy languages to express different access control restrictions, as long as policies represent a deterministic automaton.

In the next chapter we will describe how authorizations could be implemented in a run-time environment to ensure that they are safe and reliable and also discuss how user views could be treated in the absence of a trusting compiled code. We also present the implementational details of our typechecker.

5. Implementation

In this chapter we discuss implementational details of our approach. We give an example how encryption could be used to ensure that authorizations are trusted and safe to use. Then, we present the typechecker implementation, including how we enable the possibility to parse a program with a generic policy language, as described in section 3.2.

The parser for the language was created with the open source parser generator Javacc [1]. This tool automatically generates a parser in Java based on a given grammar. It also allows to insert custom code to create additional information, particularly to create abstract syntax trees (ASTs) for each instruction.

5.1 Securing Authorizations via Encryption

In order to achieve a reliable and safe access control we need to make sure that authorizations are encrypted somehow, since they are considered a first class value. To that end, in this section we purpose a design for a run-time environment capable of handling authorizations.

The run-time environment (RE) is a secure space where all objects are located, and is the trusted computing base of all process running in that environment (figure 5.1).

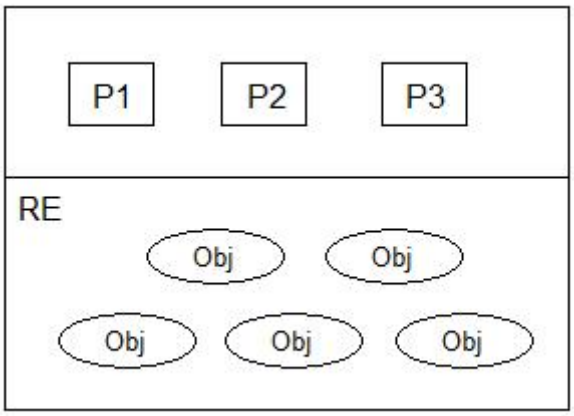

Figure 5.1 Run-time environment structure.

So, when a process creates a new object, it must ask the RE to create that object. In its turn, the RE returns a new user view with the location for that object and the maximum class policy (figure 5.2), as we have seen in the previous chapters.

Now the process can create authorizations. Since the RE is trusted by all processes, it is it who is capable of creating authorizations that can be trusted by all of them. As such, when a process wants to create an authorization, it will ask the RE to do it, sending it the user view

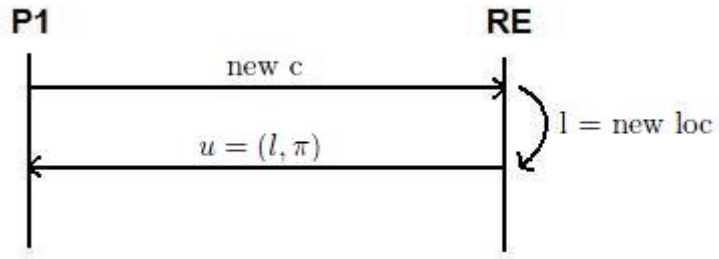

Figure 5.2 Interaction between a process and the RE when a new object is created.

and the desired policy (figure 5.3). In its turn, the RE creates a new authorization with the following format: $a = \{l, \pi', \{ts\}_{priv}\}_{pub}$, returning it to the process. The purpose for those two encryptions is to guarantee authorization integrity and authenticity. The RE encrypts some information (a timestamp, for example) with its private key. That way, when receiving an authorization, a RE can verify if it was created by it simply by decrypting that information, guaranteeing authenticity. Then, it encrypts the object location (from the user view) together with the policy and the previous cipher using its public key. That way, only the RE is able to decrypt that authorization, guaranteeing its integrity.

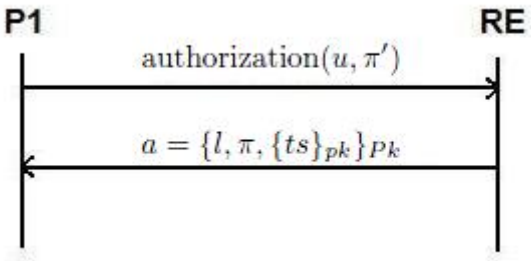

Figure 5.3 Interaction between a process and the RE when an authorization is created.

Both this encryptions ensure that only the RE can handle authorizations to its objects. Hence, processes can freely send authorizations to other processes to whom they authorized the access.

To apply an authorization, a process must also ask the RE to do so, sending the user view, the authorization and the desired policy (figure 5.4). The RE, in its turn, will first try to decrypt the authorization with its private key to gets its values (location, policy and ciphered timestamp). If successful, it will try to decrypt the ciphered timestamp with its public key. Finally, it will compare the locations and policies as we have described, returning OK or KO. If any decryption fails, the RE returns KO, since the authorization might not have been created by it.

As we assume that processes' programs are typechecked with our type system, method calls

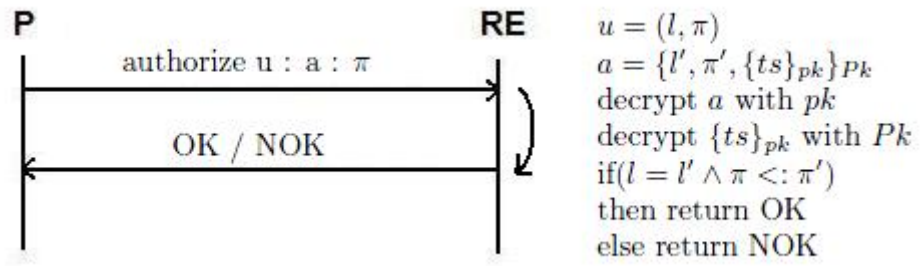

Figure 5.4 Interaction between a process and the RE when an authorization is applied.

can be executed without any run-time check, since the type system guarantees that every call follows an authorized policy. In the next section we discuss how we could implement our solution if this assumption can not be considered.

To conclude, we believe that this design can be a source of inspiration, if not a starting point, to an actual implementation of a run-time environment capable of securing trustworthy authorizations.

5.2 Trusted Code Translation

5.2.1 The Problem

Type systems, or any static analysis, are implemented by a tool (compiler with typechecker) that is assumed to be used to verify every code written in our language and transform the code into other language for it to be executed (Java, for example). However, the run-time environment should not accept any code without some sort of verification to assert that it is trusted, that is, it complies with the type system rules, since anyone could change it. To that end, and as we have seen in section 2.1.4, we could use other language-based techniques to make the code more reliable: in-lined execution monitors [20, 19, 21] and certifying compilers [32] with proof carrying code [43, 44].

On the other hand, these techniques might not be possible to integrate on an existing environment [2]. As such, we need a tool for transforming a code written with our language to another target language, guaranteeing the same properties that of our type system.

In our solution, the problem resides on method calls. Any method call is considered safe if it typechecks, since it will be an authorized call. If a simple compiler using our typechecker transformed the given code into Java code, for example, implementing the user views as simple objects containing the protected object and the policy, any user would be able to temper with every object's method calls in the Java code, since it would not be typechecked again with our typechecker. So, we need an implementational technique to still be able to guarantee that method can only be called if they follow the current policy, even if they where tempered with

in the target language.

5.2.2 The Proposed Solution

We propose the user views be transform into reference monitors, where calls are controlled according to the state of the policy.

Monitors are a mechanism used to prevent malicious accesses to protected data, wrapping it with run-time checks to verify security concerns. If the verification fails, then the access is denied. As we can see, these technique is similar to our user views. Both have a protected object and a policy that restricts the access to that object. As for user views, objects can only be accessed through a monitor.

So, user views could be transformed into reference monitors. A user view's policy would be implemented with a state transition system (an automaton) and every time a method was called, the monitor would check if the method name was available in the current policy state. Since monitors would only created by the run-time environment (in the design shown in the previous section), any modifications to method calls in the transformed code would be detected by the monitor.

This solution has the same properties of our type system, since it performs the same verifications, though introducing run-time checks for each method call.

5.3 TypeChecker

The parser for our language was created with the parser generator Javacc [1]. The given grammar has custom code to create an abstract syntax tree for each syntactic construction once the parser is executed.

Our typechecking algorithm [1] takes advantage of that. The abstract trees will be organized hierarchically, from the main node (the program) to the leafs (expressions with no subexpressions). So, we use an inductive algorithm, much like the typing rules, calling the typechecker every time an expression needs to be typechecked. Each instruction will be represented by a java class implementing an interface that has the typecheck method:

```
public interface IASTExpression {
  public IType typeCheck(Environment<IType> env,
    PiLanguage pil, boolean hasEffect) throws Exception;
}
```

As we can see, the typechecker method needs three arguments - an environment, a policy language and a boolean value -, returns a type if the expression is well-typed and may throw an exception a typing error occurs. The following sections 5.3.1 and 5.3.2 describe the implementation of types and the typing environment, to be used as the result of the typecheck and its first

[1] Complete implementation available at `http://ctp.di.fct.unl.pt/PLASTIC/`

argument, and how we enable a generic policy language, for the second argument. The boolean value is a simple implementational detail for the typing rules of *id*, *this* and *field access* (for short, *name*). If it is true, then the typing will have an effect (rules [T - *name* - 2]), otherwise there will be no changes (rules [T - *name* - 1]).

5.3.1 Types and Typing Environment

We developed a generic environment that associates strings to a generic type:

```
public class Environment<T extends Copiable<T> > {
  private Map<String, T> ids;
  protected Environment<T> previous;

  public void setPrev(Environment<T> prev){...}
  public Environment<T> BeginScope(){...}
  public void Assoc(String id, T value){...}
  public T Find(String id)throws Exception{...}
  public Environment<T> EndScope(){...}
  public Environment<T> Copy() {...}
  public boolean equals(Object obj){...}
}
```

The environment is implemented as a linked list, where each node is a program scope. So, each time a new scope is needed, as in the variable binding or method body, a new environment (node) is created linked with the previous environment using the `BeginScope()` method. Each new variable is then added to the new environment, using the `Assoc()` method, while the program is typechecked and, when it finishes, the environment is discarded, leaving the previous environments untouched. To get the type of an element, we use the `Find()` method, that looks for the element at each environment node. If the element was not found it means it was undeclared and an exception is thrown.

As we have seen, the typing environments will contain elements associated with types. To that end, we created an interface for types called *IType* that every type must implement:

```
public interface IType
   extends Copiable<IType>
{ }
```

As such, we created a type class for each type considered in section 4.4, with minor modifications:

- IntType, to represent the integer type;

- AuthType, to represent the authorization type;
- UserType, to represent the user type and containing the object type information (fields and methods);
- ClassType, to represent the class information;
- MethodType, to represent the method type;
- OKType, to state that a class was well-typed.

Please refer to the implementation [1] for more details.

5.3.2 Enabling Generic Policy Languages

One of the interesting properties of this work is the possibility of using different languages to define the policies to objects. To allow this possibility, when expecting a policy the parser reads the whole policy as a token and passes it to a second parser to parse it, as following:

```
Policy pi(): { String s = ""; } {
  <LB> { token_source.SwitchTo(PI) ; }
  ( <TEXT> {s+=token.image;} )*
  <RBPI> {return pi.parse(s);}
}
```

This is achieved using a feature from Javacc - lexical states. One can assign tokens only to be considered when the parser is on the respective lexical state. Hence, we defined a lexical state PI on which any character can be read, as following:

```
<PI> TOKEN : { < RBPI: "}" > : DEFAULT | < TEXT: ~[] > }
```

To enter that state, we force the parser to change states with the `token_source.SwitchTo()` method when a brace is opened and a policy is expected. All characters will be part of the token until the brace is closed. The limitation of this approach is the impossibility to use braces in any policy language, since it would interfere with this management, though we consider this limitation to be irrelevant.

Therefore, when parsing the code it is necessary to provide a parser for the desired policy language. To do so, we created an interface *PiLanguage* that must be implemented by any parser for policy languages:

```
public interface PiLanguage {
  Policy parse(String pi);
  Policy newFullAccess();
  Policy newEmptyAccess();
}
```

This interface has the method *parse* that receives the coded policy to parse. The result of this operation is an object instance of a class implementing the interface *Policy*:

```
public interface Policy {
  boolean isValidWith(List<String> methodNames);
  boolean isSubsetOf(Policy policy);
  Policy step(String methName);
}
```

This interface provides the three necessary properties for a policy language to be used with the developed type system (see 3.2), namely: *isValidWith* (validation property), *isSubsetOf* (subset property) and *step* (progress property).

Thus, to create a language for policies compatible with the type system, one must extend the respective parser with the interface PiLanguage, implementing the *parser* method, and create a class that extends the interface Policy, implementing the necessary properties.

The developed typechecker [1] has the implementation of the Set of Methods policy language described in section 3.2.1.

6. Closing Remarks

In this dissertation we have presented a type system which guarantees a proper use of object in accordance to a policy. Not only that, we allow the dynamic modification of policies using authorizations. To keep track of this modifications, we defined user views as enhanced object references that contain the current policy for that object. This approach to typing access control, by moving access control policies from the resources to the user views, does not seem to have been much explored before this work. Additionally, we allow a generic policy language to be used to define policies.

We have shown a formalization of this type system using typing rules, covering the most important constructions found in any object-oriented language: class declaration, object creation, field manipulation, method calls, and variable binding. Some typing rules use abstract functions provided by the policy language as its premises. Therefore, our typing system is somewhat generic concerning the expressiveness of policies.

We devised and prove a type soundness theorem that states well-typed programs do not get stuck in error states, particularly a unauthorized call to a method, since we consider an object usage to be its methods calls.

As a proof-of-concept, we created a prototype typechecker using the presented language. We described how we allow a generic policy language to parse the policies and how the typechecking is performed. We wrote a few small examples to be verified with the typechecker to demonstrate some properties that our solution possess.

We wrote a larger example, but with some extensions to the presented language. This does not invalidate the example, since these extensions are minor details to make the language more user friendly, though making it impossible to be verified with the current version of the typechecker.

From our point of view, we have fulfilled all objectives as we created a type and effect system capable of detecting unauthorized or unexpected method calls and a working prototype typechecker for the presented language and type system. We also discussed how a run-time environment should consider authorizations to make them safe and reliable. So, we think this work serves as a good starting point for a larger study of the approach, namely how it can be integrated with conventional object-oriented languages, like Java, and in presence of concurrency.

6.1 Validation

This work has been validated by two components. First and foremost, the proof of type safety, specially the proof of the subject reduction theorem, proving that the type system complies with its goals - the detection of unauthorized method calls. Second, the proof-of-concept implementation of the typechecking algorithm, the prototype typechecker, allows the validation of some examples, both well-typed and ill-typed.

6.2 Future Work

In this final section we present some improvements, research problems and challenges for future work based on this dissertation. We rank them by how important they are.

A run-time implementation, as suggested in section 5.1, should be the first priority in order to evaluate our approach in comparison to other approaches. Our language could also be improved with some extensions for it to be used to write richer programs and, as we are using an object-oriented language, inheritance is inevitable and should be integrated in the language. Then, we could reason how concurrency could be considered by our approach.

Run-time Implementation: In section 5.1 we describe a possible implementation of a run-time environment capable of making authorizations secure and reliable through encryption. This brief description is by no means sufficient to accept it as the final solution, since it may not cover all problems related to the implementation. However, as we discussed the main issues and proposed solutions, it can be used as a starting point to apply the desired properties.

Language Extensions: Although our language is sufficient to show the potentials of the approach design in this dissertation, for it to be properly used like any other object-oriented language, we must extend it with other traditional construction: code branch, loops, multiple parameter methods and exceptions. These extensions must be studied carefully when dealing with user views, though we believe that no major problems would arise.

Inheritance: We left out inheritance from the beginning of this work due to our belief that it would not affect our solution, only adding an unnecessary layer of complexity. However, some of the challenges required the addition of the maximal class policy. So, this class-dependent property would have to be considered when performing inheritance. This remains an open issue and an interesting topic of research.

Concurrency: We design this type system for sequential programming. However, on a more realistic setting, we wonder how user views and authorizations should behave in concurrent programming. The possibility of shared user views is a big challenge for our approach, as well as the security concerns that authorization handling implicitly have on a concurrent environment.

Other improvements: There are improvements that could be interesting to consider for a more powerful and flexible type system, but we considered insignificant for this thesis goals.

As we have seen by rule [T - id], we could have the possibility to decompose a policy into two policies, allowing, for example, authorized parallel object usage.

The rule [T - Authorize] allows an authorization to be applied to a user view whose policy is in any state. The programmer could want to be sure that the policies were strictly followed, restricting the applicability of authorize to user views whose policies

where *stable* (empty or in the initial state of a protocol). This could even be a flag for the typechecker to allow both demands.

When developing the type system, we assumed that all objects are protected. However, this could not be what the programmer wanted, since it would imply the use of authorizations for every other objects. To consider this possibility, both the language and type system must be modified, though without consequences for the current results.

A . Selected Proofs

A.1 Subject Reduction

Theorem A.1.1. *Subject Reduction. Let Δ and Δ'' be a typing environments, e an expression such that e is closed in Δ and T a type.*

$$\begin{array}{ll} \textit{if} & \Delta \vdash e : T \mapsto \Delta'' : e \textit{ is closed in } \Delta \\ \textit{and} & \exists\, S \,:\, \Delta \vdash S \\ \textit{and} & < S \,;\, e > \longrightarrow < S' \,;\, e' > \\ \\ \textit{then} & \exists\, \Delta', \Delta'' \,:\, \Delta' \vdash S',\ \Delta \subseteq^{\pi} \Delta',\ \Delta'' \subseteq \Delta''' \\ \textit{and} & \Delta' \vdash e' : T \mapsto \Delta'''. \end{array}$$

Proof. By rule induction on the derivation of $< S \,;\, e > \longrightarrow < S' \,;\, e' >$.

Case [Object Creation]

We have

- $< S \,;\, \textit{new } c > \longrightarrow < S' \,;\, u >$

  $S' = S \uplus \{l \to \{F_s \ M_s\}, u \to (l,\ \pi)\}$

- $\Delta \vdash S \Rightarrow M \vdash M_s \wedge \exists\, c : [F \ \ M \ \ \pi] \in \Delta$

- $\Delta \vdash \textit{new } c : M\{\pi\} \mapsto \Delta''$

  $\Delta'' = \Delta$

We also have by [T - id - 2]

- $\Delta, u : M\{\pi\} \vdash u : M\{\pi\} \mapsto \Delta, u : M\{\emptyset\}$

So, let $\Delta' = \Delta, u : M\{\pi\} \ \Rightarrow \Delta \subseteq^{\pi} \Delta'$ and $\Delta' \vdash S'$
and let $\Delta''' = \Delta'', u : M\{\emptyset\} \ \Rightarrow \Delta'' \subseteq \Delta'''$.

By Lemma 4.4.2(weakening), we get $\Delta' \vdash u : M\{\pi\} \mapsto \Delta'''$.

Case [Field Access]

We have

- $< S\ ;\ u.fd > \longrightarrow < S'\ ;\ u' >$
  $S' = S[u_t.fd \leftarrow (l',\ \emptyset)] \uplus \{u' \rightarrow (l',\ \pi)\}$
  $u \rightarrow (l,\ \pi_1) \in S$
  $l \rightarrow \{F_s\ \ M_s\} \in S$
  $fd \rightarrow (l',\ \pi) \in F$
  $l' \rightarrow \{F'_s\ \ M'_s\} \in S$
- $\Delta \vdash S \Rightarrow M \vdash M_s \wedge M' \vdash M'_s$
- $\Delta \vdash u.fd : M'\{\pi\} \mapsto \Delta[u.fd \leftarrow \emptyset]$
  $\Delta'' = \Delta[u.fd \leftarrow \emptyset]$
  $u : [F\ \ M]\{\pi_1\} \in \Delta$
  $fd : M'\{\pi\} \in F$

Let u' be typed as follows, by [T - id - 2]:

- $u' : M'\{\pi\} \vdash u' : M'\{\pi\} \mapsto u' : M'\{\emptyset\}$

So, let $\Delta' = \Delta[u.fd \leftarrow \emptyset], u' : M'\{\pi\} \ \Rightarrow \Delta \subseteq^{\pi} \Delta'$ and $\Delta' \vdash S'$ and let $\Delta''' = \Delta'', u' : M'\{\emptyset\} \ \Rightarrow \Delta'' \subseteq \Delta'''$.

By Lemma 4.4.2(weakening), we get $\Delta' \vdash u' : M'\{\pi\} \mapsto \Delta'''$.

Case [Field Update]

We have

- $< S\ ;\ u.fd = e > \longrightarrow < S'\ ;\ u.fd = e' >$
  $< S\ ;\ e > \longrightarrow < S'\ ;\ e' >$
- $\Delta \vdash S$
- $\Delta \vdash u.fd = e : M\{\emptyset\} \mapsto \Delta''[u.fd \leftarrow \pi_e]$
  $\Delta \vdash e : M\{\pi_e\} \mapsto \Delta''$

By the induction hyphothesis, we know that

- $\exists \Delta', \Delta'''\ :\ \Delta \subseteq^{\pi} \Delta', \Delta' \vdash S', \Delta'' \subseteq \Delta'''$
  $\Delta' \vdash e' : M\{\pi_e\} \mapsto \Delta'''$

So, by [T - Field Update]

- $\Delta' \vdash u.fd = e' : M\{\emptyset\} \mapsto \Delta'''[u.fd \leftarrow \pi_e]$

Case [Field Update Base]

We have

- $< S\ ;\ u.fd = u' > \longrightarrow < S'\ ;\ u' >$
  $S' = S[u.fd \leftarrow (l',\ \pi), u \leftarrow (l',\ \emptyset)]$
  $u \rightarrow (l,\ \pi_1) \in S$
  $l \rightarrow \{F_s\ \ M_s\} \in S$
  $fd \in Dom(F_s)$
  $u' \rightarrow (l',\ \pi) \in S$
  $l' \rightarrow \{F'_s\ \ M'_s\} \in S$
- $\Delta \vdash S \Rightarrow F \vdash F_s \wedge M \vdash M_s \wedge F' \vdash F'_s \wedge M' \vdash M'_s$
- $\Delta \vdash u.fd = u' : M'\{\emptyset\} \mapsto \Delta[u' \leftarrow \emptyset, u.fd \leftarrow \pi]$
  $\Delta'' = \Delta[u' \leftarrow \emptyset, u.fd \leftarrow \pi]$
  $\Delta \vdash u' : M'\{\pi\} \mapsto \Delta[u' \leftarrow \emptyset]$
  $u : [F\ \ M]\{\pi_1\} \in \Delta$
  $fd : M'\{\pi'\} \in F$
  $u' : M'\{\pi\} \in \Delta$

We also have by [T - id - 1]

- $\Delta[u' \leftarrow \emptyset] \vdash u' : M'\{\emptyset\} \mapsto \Delta[u' \leftarrow \emptyset]$

So, let $\Delta' = \Delta[u' \leftarrow \emptyset, u.fd \leftarrow \pi] \ \Rightarrow \Delta \subseteq^{\pi} \Delta'$ and $\Delta' \vdash S'$
and let $\Delta''' = \Delta'' \ \Rightarrow \Delta'' \subseteq \Delta'''$.

By Lemma 4.4.2(weakening), we get $\Delta' \vdash u' : M'\{\emptyset\} \mapsto \Delta'''$.

Case [Variable Binding]

We have

- $< S\ ;\ let\ x = e_1\ in\ e_2 > \longrightarrow < S'\ ;\ let\ x = e_3\ in\ e_2 >$
  $< S\ ;\ e_1 > \longrightarrow < S'\ ;\ e_3 >$

- $\Delta \vdash S$
- $\Delta \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : M_2\{\pi_2\} \mapsto \Delta''$
  $\Delta \vdash e_1 : M_1\{\pi_1\} \mapsto \Delta_1$
  $\Delta_1, x : M_1\{\pi_1\} \vdash e_2 : M_2\{\pi_2\} \mapsto \Delta'', x : M_1\{\pi_1'\}$

By the induction hyphothesis, we know that

- $\exists \Delta', \Delta_1''' \ :\ \Delta_1' \vdash S', \Delta \subseteq^{\pi} \Delta', \Delta_1 \subseteq \Delta_1'''$
  $\Delta' \vdash e_3 : M_1\{\pi_1\} \mapsto \Delta_1'''$

By Lemma 4.4.2(weakening), we have

- $\Delta_1''', x : M_1\{\pi_1\} \vdash e_2 : M_2\{\pi_2\} \mapsto \Delta''', x : M_1\{\pi_1'\}$
  $\Delta''' \subseteq \Delta''$

By applying [T - Variable Binding], we get

- $\Delta' \vdash \mathit{let}\ x = e_3\ \mathit{in}\ e_2 : M_2\{\pi_2\} \mapsto \Delta'''$.

Case [Variable Binding Base]

We have

- $< S\ ;\ \mathit{let}\ x = u\ \mathit{in}\ e > \longrightarrow < S'\ ;\ e\{^{u_2}/_{x}\} >$
  $S' = S[u \leftarrow (l,\ \emptyset)] \uplus \{u_2 \rightarrow (l,\ \pi)\}$
  $u \rightarrow (l,\ \pi) \in S$
  $l \rightarrow \{F_{S1}\ \ M_{S1}\} \in S$
  $u_2 \notin S$
- $\Delta \vdash S \Rightarrow M_1 \vdash M_{S1}$
- $\Delta \vdash \mathit{let}\ x = u\ \mathit{in}\ e : M_2\{\pi_2\} \mapsto \Delta''$
  $\Delta \vdash u : M_1\{\pi\} \mapsto \Delta_1$
  $\Rightarrow \Delta \subseteq^{\pi} \Delta_1$
  $\Delta_1, x : M_1\{\pi\} \vdash e : M_2\{\pi_2\} \mapsto \Delta'', x : M_1\{\pi'\}$

We also have by [T - id - 2]

- $\Delta \vdash u : M_1\{\pi\} \mapsto \Delta[u \leftarrow \emptyset]$
  as such, $\Delta_1 = \Delta[u \leftarrow \emptyset]$

Let u_2 be a new user view typed as follows:

- $u_2 : M_1\{\notin\}\Delta_1$

By Lemma 4.4.1 (substitution lemma), we have:

- $\Delta_1, u_2 : M_1\{\pi\} \vdash e\{^{u_2}/_x\} : M_2\{\pi_2\} \mapsto \Delta'', u_2 : M_1\{\pi'\}$

So, let $\Delta' = \Delta_1, u_2 : M_1\{\pi\} \;\; \Rightarrow \Delta \subseteq^{\pi} \Delta'$ and $\Delta' \vdash S'$
and let $\Delta''' = \Delta'', u_2 : M_1\{\pi'\} \;\; \Rightarrow \Delta'' \subseteq \Delta'''$.

By Lemma 4.4.2(weakening), we get $\Delta' \vdash e\{^{u_2}/_x\} : M_2\{\pi_2\} \mapsto \Delta'''[u \leftarrow \emptyset]$.

Case [Method Call]

We have

- $< S \; ; \; u.m(e) > \longrightarrow < S' \; ; \; u.m(e') >$
  $< S \; ; \; e > \longrightarrow < S' \; ; \; e' >$
- $\Delta \vdash S$
- $\Delta \vdash u.m(e) : M_1\{\emptyset\} \mapsto \Delta''[u \leftarrow \pi]$
  $\Delta \vdash e : M_2\{\emptyset\} \mapsto \Delta''$
  $u : M_x\{\pi'_x\} \in \Delta''$
  $\pi'_x \xrightarrow{m} \pi$

By the induction hyphothesis, we know that

- $\exists \Delta', \Delta''' \; : \; \Delta \subseteq^{\pi} \Delta', \Delta' \vdash S', \Delta'' \subseteq \Delta'''$
  $\Delta' \vdash e' : M_2\{\emptyset\} \mapsto \Delta'''$
  $u : M_x\{\pi'_x\} \in \Delta'''$
  $\pi'_x \xrightarrow{m} \pi$

So, applying [T - Method Call] we get

- $\Delta' \vdash u.m(e') : M_1\{\emptyset\} \mapsto \Delta'''[u \leftarrow \pi]$

Case [Method Call - Base]

We have

- $< S\ ;\ u_1.m(u_2) > \longrightarrow < S'\ ;\ e_m\{^{u_3}/_{x}\}\{^{u_t}/_{this}\} >$

  $S' = S[u_1 \leftarrow \pi'] \uplus \{u_3 \rightarrow (l_u,\ \emptyset), u_t \rightarrow (l,\ *)\}$

  $u_1 \rightarrow (l,\ \pi) \in S \qquad \pi \xrightarrow{m} \pi'$
  $l \rightarrow [F_s\ \ M_s] \in S \qquad u_2 \rightarrow (l_u,\ \pi_u) \in S$
  $m(x) = e_m \in M_s \qquad l_u \rightarrow [F_{s2}\ \ M_{s2}] \in S_2$

- $\Delta \vdash S \Rightarrow M \vdash M_s \wedge M_1 \vdash M_{s1} \wedge M_2 \vdash M_{s2}$

- $\Delta \vdash u_1.m(u_2) : M_1\{\emptyset\} \mapsto \Delta_1[u_1 \leftarrow \pi']$

  $\Delta'' = \Delta_1[u_1 \leftarrow \pi']$

  $\Delta \vdash u_2 : M_2\{\emptyset\} \mapsto \Delta_1$

  $u_1 : M\{\pi\} \in \Delta_1$

  $\pi \xrightarrow{m} \pi'$

  $m : M_1(M_2) \in M$

We know by [T - id - 1]

- $\Delta \vdash u_2 : M_2\{\emptyset\} \mapsto \Delta$

  so, $\Delta_1 = \Delta$

By [T-Class], let $\Delta_m = \Delta_c, this : [F\ \ M]\{*\}\emptyset, x : M_2\{\}$

- $\Delta_m \vdash e_m : M_1\{\emptyset\} \mapsto \Delta_m$

Let u_t and u_3 be new user views typed as follows:

- $u_t : [F\ \ M]\{*\} \notin \Delta_m \cup \Delta$
- $u_3 : M_2\{\emptyset\} \notin \Delta_m \cup \Delta$

By Lemma 4.4.1 (substitution lemma):

- let $\Delta_m\{^{u_3}/_{x}\}\{^{u_t}/_{this}\} = \Delta_c, u_t : [F\ \ M]\{*\}, u_3 : M_2\{\emptyset\}$

  $\Delta_m\{^{u_3}/_{x}\}\{^{u_t}/_{this}\} \vdash e_m\{^{u_3}/_{x}\}\{^{u_t}/_{this}\} : M_1\{\emptyset\} \mapsto \Delta_m\{^{u_3}/_{x}\}\{^{u_t}/_{this}\}$

So, let $\Delta' = \Delta[u_1 \leftarrow \pi'], \Delta_c, u_t : [F\ \ M]\{*\}, u_3 : M_2\{\emptyset\} \ \Rightarrow \Delta \subseteq^{\pi} \Delta'$ and $\Delta' \vdash S'$ and let $\Delta''' = \Delta'', \Delta_c, u_t : [F\ \ M]\{*\}, u_3 : M_2\{\emptyset\} \ \Rightarrow \Delta'' \subseteq \Delta'''$.

By Lemma 4.4.2(weakening), we get $\Delta' \vdash e_m\{^{u_3}/_{x}\}\{^{u_t}/_{this}\} : M_1\{\emptyset\} \mapsto \Delta'''$.

Case [Authorization]

We have

- $< S\ ;\ authorization(u,\pi) > \longrightarrow < S'\ ;\ a >$

  $S' = S \uplus \{a \rightarrow [l,\ \pi]\}$

  $u \rightarrow (l,\ \pi_u) \in S$

  $l \rightarrow \{F_s\ \ M_s\} \in S$

  $\pi <: \pi_u$

- $\Delta \vdash S \Rightarrow M \vdash M_s$

- $\Delta \vdash authorization(u,\pi) : Auth \mapsto \Delta''$

  $\Rightarrow \Delta'' = \Delta$

  $u : M\{\pi_u\} \in \Delta$

  $\pi <: \pi_u$

Let $\Delta' = \Delta, a : Auth \ \Rightarrow \Delta \subseteq^{\pi} \Delta'$ and $\Delta' \vdash S'$
and let $\Delta''' = \Delta'', a : Auth \Rightarrow \Delta'' \subseteq \Delta'''$.

By Lemma 4.4.2(weakening), we get $\Delta' \vdash a : Auth \mapsto \Delta'''$.

Case [Authorize OK]

We have

- $< S\ ;\ \begin{array}{l} authorize\ u : a \\ case\ \pi : \{e_1\} \\ case\ error: \{e_2\} \end{array} > \longrightarrow < S'\ ;\ e_1 >$

  $S' = S \uplus \{u' \rightarrow (l,\ \pi)\}$

  $u \rightarrow (l,\ \pi_u) \in S$

  $l \rightarrow \{F_s\ \ M_s\} \in S$

  $S(a) \vdash [l,\ \pi]$

- $\Delta \vdash S \Rightarrow M \vdash M_s$

- $\Delta \vdash \begin{array}{l} authorize\ u : a \\ case\ \pi : \{e_1\} \\ case\ error: \{e_2\} \end{array} : T \mapsto \Delta''$

  $\Delta[u \leftarrow \pi] \vdash e_1 : T \mapsto \Delta''$

  $a : Auth \in \Delta$

  $u : M\{\pi_u\} \in \Delta$

  $u : M\{\pi'_u\} \in \Delta''$

So, let $\Delta' = \Delta[u \leftarrow \pi] \Rightarrow \Delta \subseteq^{\pi} \Delta'$ and $\Delta' \vdash S'$
and let $\Delta''' = \Delta'' \Rightarrow \Delta'' \subseteq \Delta'''$.

We immediately get $\Delta' \vdash e_1 : T \mapsto \Delta'''$.

Case [Authorize NOK] We have

- $< S ; \begin{array}{l} \textit{authorize } u : a \\ \textit{case } \pi : \{e_1\} \\ \textit{case error: } \{e_2\} \end{array} > \longrightarrow < S' ; e_2 >$

  $S' = S$

  $u \rightarrow (l, \pi_u) \in S$

  $l \rightarrow \{F_s \ M_s\} \in S$

  $S(a) \not\vdash [l, \pi]$

- $\Delta \vdash S \Rightarrow M \vdash M_s$

- $\Delta \vdash \begin{array}{l} \textit{authorize } u : a \\ \textit{case } \pi : \{e_1\} \\ \textit{case error: } \{e_2\} \end{array} : T \mapsto \Delta''$

  $\Delta \vdash e_2 : T \mapsto \Delta''$

  $a : Auth \in \Delta$

  $u : M\{\pi_u\} \in \Delta$

So, let $\Delta' = \Delta \Rightarrow \Delta \subseteq^{\pi} \Delta'$ and $\Delta' \vdash S'$
and let $\Delta''' = \Delta'' \Rightarrow \Delta'' \subseteq \Delta'''$.

We immediately get $\Delta' \vdash e_2 : T \mapsto \Delta''$

□

A.2 Progress

Theorem A.2.1. *Progress. Let Δ and Δ' be a typing environments, e an expression such that e is closed in Δ and T a type.*

if $\Delta \vdash e : T \mapsto \Delta' : e$ *is closed in* Δ

then $\exists S : \Delta \vdash S$
and $< S ; e > \longrightarrow < S' ; e' >$
or e *is a value.*

Proof. By rule induction on the typing derivation.

Case [T - Object Creation]

We have

$$\frac{c : [F \;\; M \;\; \pi] \in \Delta}{\Delta \vdash new\ c : M\{\pi\} \mapsto \Delta}$$

Let S be a store such that $\Delta \vdash S$. Then, we know that there is a class named c in the store, $c \to \{F_s \;\; M_s \;\; \pi\} \in S$. So, we have the necessary conditions to apply rule [R - Object Creation], validating the conclusion.

Case [T - id - 1] & [T - id - 2]

We have

[T - id - 1] [T - id - 2]

$$\frac{id : M\{\pi\} \in \Delta}{\Delta \vdash id : M\{\emptyset\} \mapsto \Delta} \qquad \frac{id : M\{\pi\} \in \Delta}{\Delta \vdash id : M\{\pi\} \mapsto \Delta[id \leftarrow \emptyset]}$$

This rules can be applied to variables or user views. So, we need to consider each case.

- *Sub-case 1*: id is a variable. This sub-case is impossible, since the expression e is closed in Δ.
- *Sub-case 2*: id is an user view. As user views are values, the conclusion is valid.

Cases [T - this - 1], [T - this - 2] and [T - Authorization Value]

This expressions are values, so the conclusion is valid.

Case [T - Field Access]

We have

$$\frac{id : [F \;\; M]\{\pi\} \in \Delta \qquad fd : M'\{\pi'\} \in F}{\Delta \vdash id.fd : M'\{\pi'\} \mapsto \Delta[id.fd \leftarrow \emptyset]}$$

Let S be a store such that $\Delta \vdash S$.
Then, we know that

- $id \to (l,\ \pi) \in S$
- $l \to \{F_s \;\; M_s\} \in S$
- $fd \to (l',\ \pi') \in F$

So, we have the necessary conditions to apply rule [R - Field Access], validating the conclusion.

Case [T - Field Update]

We have

$$\frac{id : [F\ \ M]\{\pi\} \in \Delta \qquad fd : M'\{\pi'\} \in F \qquad \Delta \vdash e : M'\{\pi_e\} \mapsto \Delta'}{\Delta \vdash id.fd = e : M'\{\emptyset\} \mapsto \Delta'[id.fd \leftarrow \pi_e]}$$

By the induction hyphothesis, we have $\exists\, S : \Delta \vdash S$ and either $< S\ ;\ e > \longrightarrow < S'\ ;\ e' >$ or e is a value. We have to consider each case.

- *Sub-case 1*: $< S\ ;\ e > \longrightarrow < S'\ ;\ e' >$. We have the necessary conditions to apply rule [R - Field Update], validating the conclusion.

- *Sub-case 2*: e is a value. Since e has type $M'\{\pi_e\}$, it is an user view ($e = u$). By $\Delta \vdash S$, we have:

  a) $id \rightarrow (l,\ \pi) \in S$

  b) $l \rightarrow \{F_s\ \ M_s\} \in S$

  c) $fd \in Dom(F_s)$

  d) $u \rightarrow (l',\ \pi_e) \in S$

  So, we have the conditions to apply rule [R - Field Update - Base], validating the conclusion.

Case [T - Variable Binding]

We have

$$\frac{\begin{array}{c}\Delta_1 \vdash e_1 : M_1\{\pi_1\} \mapsto \Delta_2 \\ \Delta_2, x : M_1\{\pi_1\} \vdash e_2 : M_2\{\pi_2\} \mapsto \Delta_3, x : M_1\{\pi_3\}\end{array}}{\Delta_1 \vdash let\ x = e_1\ in\ e_2 : M_2\{\pi_2\} \mapsto \Delta_3}$$

By the induction hyphothesis, we have $\exists\, S : \Delta \vdash S$ and either $< S\ ;\ e_1 > \longrightarrow < S'\ ;\ e_1' >$ or e is a value. We have to consider each case.

- *Sub-case 1*: $< S\ ;\ e_1 > \longrightarrow < S'\ ;\ e_1' >$. We have the necessary conditions to apply rule [R - Variable Binding], validating the conclusion.

- *Sub-case 2*: e is a value. Since e has type $M_1\{\pi_1\}$, it is an user view ($e = u$). By $\Delta \vdash S$, we have $u \rightarrow (l,\ \pi) \in S$. So, we have the conditions to apply rule [R - Variable Binding - Base], validating the conclusion.

Case [T - Method Call]

We have

$$\frac{\begin{array}{ll}\Delta \vdash e : M_2\{\emptyset\} \mapsto \Delta' & m : M_1(M_2) \in M_x \\ id : M\{\pi_x\} \in \Delta & \pi'_x \overset{m}{\to} \pi \\ id : M\{\pi'_x\} \in \Delta' & \end{array}}{\Delta \vdash id.m(e) : M_1\{\emptyset\} \mapsto \Delta'[x \leftarrow \pi]}$$

By induction hyphothesis, we have $\exists\, S : \Delta \vdash S$ and either $< S\ ;\ e > \longrightarrow < S'\ ;\ e' >$ or e is a value. We have to consider each case.

- *Sub-case 1*: $< S\ ;\ e > \longrightarrow < S'\ ;\ e' >$. We have the necessary conditions to apply rule [R - Method Call], validating the conclusion.
- *Sub-case 2*: e is a value. Since e has type $M_2\{\emptyset\}$, it is an user view ($e = u_2$). By $\Delta \vdash S$, we have:

  a) $id \to (l,\ \pi_x) \in S$

  b) $l \to \{F_s\ \ M_s\} \in S$

  c) $m(x) = e \in M_s$

  d) $u_2 \to (l',\ \emptyset) \in S$

  So, we have the conditions to apply rule [R - Method Call - Base], validating the conclusion.

Case [T - Authorization]

We have

$$\frac{id : M\{\pi_x\} \in \Delta \qquad \pi <: \pi_x \qquad M \vdash \pi}{\Delta \vdash authorization(id, \pi) : Auth \mapsto \Delta}$$

Let S be a store such that $\Delta \vdash S$. Then, we know that there is an user view named id in the store, $id \to (l,\ \pi_u) \in S$. So, we have the necessary conditions to apply rule [R - Authorization], validating the conclusion.

Case [T - Authorize]

We have

$$\frac{\begin{array}{ll}id : M\{\pi_x\} \in \Delta & \Delta[id \leftarrow \pi] \vdash e_1 : T \mapsto \Delta' \\ a : Auth \in \Delta & \Delta \vdash e_2 : T \mapsto \Delta' \\ M \vdash \pi & \end{array}}{\Delta \vdash \begin{array}{l} authorize\ id : a \\ case\ \pi :\ \{e_1\} \\ case\ error:\ \{e_2\} \end{array} : T \mapsto \Delta'}$$

Let S be a store such that $\Delta \vdash S$. Then, we know that there is an user view named *id* in the store, $id \rightarrow (l,\ \pi_x) \in S$, and an authorization named a, $a \rightarrow [l_a,\ \pi_a]$.

If $S(a) \vdash (l, \pi)$ then we have the necessary conditions to apply rule [R - Authorize OK], validating the conclusion; else, we know that $S(a) \nvdash (l, \pi)$ and so we have the necessary conditions to apply rule [R - Authorize NOK], validating the conclusion.

□

Bibliography

[1] Javacc homepage, July 2009. https://javacc.dev.java.net/.

[2] Martín Abadi. Protection in programming-language translations. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 868–883. Springer-Verlag, 1998.

[3] Martín Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5):749–786, 1999.

[4] Martín Abadi. Logic in access control. In *Proceedings of the 18th Annual Symposium on Logic in Computer Science (LICS'03*, pages 228–233. IEEE Computer Society Press, 2003.

[5] Martín Abadi. On access control, data integration and their languages, 2004. Computer Systems: Theory, Technology and Applications, A Tribute to Roger Needham Springer-Verlag.

[6] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL 2008*, 2008.

[7] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst.*, 15(4):706–734, 1993.

[8] Anindya Banerjee. Representation independence, confinement and access control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 166–177. ACM Press, 2002.

[9] Anindya Banerjee and David A. Naumann. A simple semantics and static analysis for java security. Technical report, 2001.

[10] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The spec# programming system: An overview. pages 49–69. Springer, 2004.

[11] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. History-based access control with local policies. In *Proc. of Foundations of Software Science and Computation Structure 2005*, pages 316–332. Springer, 2005.

[12] Massimo Bartoletti, Pierpaolo Degano, Gian Luigi Ferrari, and Roberto Zunino. Types and effects for resource usage analysis. In *Proc. of Foundations of Software Science and Computation Structure 2007, LNCS*, pages 32–47. Springer, 2007.

[13] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 17–32, Washington, DC, USA, 2008. IEEE Computer Society.

[14] Michele Bugliesi, Dario Colazzo, Silvia Crafa, and Università Ca Foscari. Type based discretionary access control. In *CONCUR'04: Concurrency Theory*, pages 225–239. Springer, 2004.

[15] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, Boca Raton, FL, 1997.

[16] Mike Chapple. *The GSEC Prep Guide: Mastering SANS GIAC Security Essentials*, chapter 1. WILEY, 2003.

[17] Avik Chaudhuri and Martin Abadi. Secrecy by typing and file-access control. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 112–123, Washington, DC, USA, 2006. IEEE Computer Society.

[18] Jason Crampton and George Loizou. A logic of access control. *The Computer Journal*, 44:2001, 2001.

[19] Ulfar Erlingsson and Fred B. Schneider. Sasi enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, pages 87–95. ACM Press, 1999.

[20] Ulfar Erlingsson and Fred B. Schneider. Irm enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[21] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, pages 32–45, 1999.

[22] David Ferraiolo and Richard Kuhn. Role-based access controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[23] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *Programming Language Design and Implementation (PLDI*, pages 219–232. ACM Press, 2000.

[24] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins, 1999.

[25] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. In *ESOP: 14th European Symposium on Programming*, pages 141–156. Springer, 2005.

[26] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 28–38, New York, NY, USA, 1986. ACM.

[27] Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:36–47, 1999.

[28] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[29] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA, 1977.

[30] Matthew Hennessy. *Algebraic Theory of Processes*. The MIT Press, Cambridge, Mass., 1988.

[31] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[32] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50:2003, 2003.

[33] Ulla Isaksen, Jonathan P. Bowen, and Nimal Nissanke. System and software safety in critical systems. Technical report, 1996.

[34] Sushil Jajodia, Pierangela Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *IEEE Symposium on Security and Privacy*, pages 31–42, 1997.

[35] Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. Aura: a programming language for authorization and audit. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 27–38, New York, NY, USA, 2008. ACM.

[36] Dexter Kozen. Language-based security. In *Mathematical Foundations of Computer Science*, pages 284–298. Springer-Verlag, 1999.

[37] Kaspersky Lab. History of malicious programs, July 2009. http://www.viruslist.com/en/viruses/encyclopedia?chapter=153280684.

[38] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, 1992.

[39] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Jml: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[40] Stewart Lee. Essays about computer security, 1999.

[41] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[42] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[43] George C. Necula. Proof-carrying code. pages 106–119. ACM Press, 1997.

[44] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. pages 333–344. ACM Press, 1998.

[45] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *ACM Computing Surveys*, pages 114–136. Springer-Verlag, 1999.

[46] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[47] Department of Defense. Trusted computer security evaluation criteria, 1985. DoD 5200.28-STD.

[48] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[49] Benjamin C. Pierce. *Advanced Topics In Types And Programming Languages*. MIT Press, November 2004.

[50] Fred B. Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead*, pages 86–101. Springer, 2000.

[51] Christian Skalka and Scott Smith. Static enforcement of security with types. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 34–45, New York, NY, USA, 2000. ACM.

[52] Christian Skalka and Scott Smith. History effects and verification. In *APLAS'04: The Second ASIAN Symposium on Programming Languages and Systems*, pages 107–128. Springer, 2004.

[53] Eugene H. Spafford. The internet worm incident. In *ESEC '89: Proceedings of the 2nd European Software Engineering Conference*, pages 446–468, London, UK, 1989. Springer-Verlag.

[54] Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *IEEE Symposium on Security and Privacy*. Society Press, 2008.

[55] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Information and Computation*, pages 162–173, 1992.

[56] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.